Detection of web vulnerabilities via model inference assisted evolutionary fuzzing

Fabien Duchene

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 Août 2006

Présentée par

Fabien Duchène

Thèse dirigée par **Professeur Roland Groz**
et co-encadrée par **Docteur Jean-Luc Richier**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Detection of Web Vulnerabilities via Model Inference assisted Evolutionary Fuzzing

Thèse soutenue publiquement le **2 Juin, 2014**,
devant le jury composé de :

M. Jean-Luc RICHIER
Chargé de Recherche, CNRS, France, Co-encadrant de thèse
M. Roland GROZ
Professeur, Grenoble INP, France, Directeur de thèse
M. Bruno LEGEARD
Professeur, Université de Franche-Comté, France, Rapporteur
M. Herbert BOS
Professeur, VU Amsterdam, Pays-Bas, Rapporteur
M. Mario HEIDERICH
Docteur, Ruhr-Universität Bochum, Allemagne, Examinateur
M. Yves DENNEULIN
Professeur, Grenoble INP, France, Examinateur

Dedication

Acknowledgements

Détection de Vulnérabilités Web par Frelatage Evolutionniste et Inférence de Modèle

Résumé: Le test est une approche efficace pour détecter des bogues d'implémentation ayant un impact sur la sécurité, c.-à-d. des vulnérabilités. Lorsque le code source n'est pas disponible, il est nécessaire d'utiliser des techniques de test en boîte noire. Nous nous intéressons au problème de détection automatique d'une classe de vulnérabilités (Cross Site Scripting alias XSS) dans les applications web dans un contexte de test en boîte noire. Nous proposons une approche pour inférer des modèles de telles applications et frelatons des séquences d'entrées générées à partir de ces modèles et d'une grammaire d'attaque. Nous inférons des automates de contrôle et de teinte, dont nous extrayons des sous-modèles afin de réduire l'espace de recherche de l'étape de frelatage. Nous utilisons des algorithmes génétiques pour guider la production d'entrées malicieuses envoyées à l'application. Nous produisons un verdict de test grâce à une double inférence de teinte sur l'arbre d'analyse grammaticale d'un navigateur et à l'utilisation de motifs de vulnérabilités comportant des annotations de teinte. Nos implémentations LigRE et KameleonFuzz obtiennent de meilleurs résultats que les scanneurs boîte noire open-source. Nous avons découvert des XSS "0-day" (c.-à-d. des vulnérabilités jusque lors inconnues publiquement) dans des applications web utilisées par des millions d'utilisateurs.

Keywords: Sécurité, Frelatage, XSS, Algorithme Evolutionniste, Inférence, Intelligence Artificielle, Applications Web

Detection of Web Vulnerabilities via Model Inference assisted Evolutionary Fuzzing

Abstract: Testing is a viable approach for detecting implementation bugs which have a security impact, a.k.a. vulnerabilities. When the source code is not available, it is necessary to use black-box testing techniques. We address the problem of automatically detecting a certain class of vulnerabilities (Cross Site Scripting a.k.a. XSS) in web applications in a black-box test context. We propose an approach for inferring models of web applications and fuzzing from such models and an attack grammar. We infer control plus taint flow automata, from which we produce slices, which narrow the fuzzing search space. Genetic algorithms are then used to schedule the malicious inputs which are sent to the application. We incorporate a test verdict by performing a double taint inference on the browser parse tree and combining this with taint aware vulnerability patterns. Our implementations LigRE and KameleonFuzz outperform current open-source black-box scanners. We discovered 0-day XSS (i.e., previously unknown vulnerabilities) in web applications used by millions of users.

Keywords: Security, Fuzzing, XSS, Evolutionary Algorithm, Inference, Artificial Intelligence, Web Applications

Contents

CHAPTER 1

Introduction

Why did I rob banks? Because I enjoyed it. I loved it.
Go where the money is...and go there often.

[Sutton & Linn 2004]

The world is a dangerous place to live ; not because of those who do evil, but because of those of the people who don't do anything about it.

[Einstein 1955]

Computer security is the cancer of the software industry. There is no money to prevent it. Only sick persons care about it, but it is generally already too late. However, everybody will have to face it someday.

[Ruff 2013b]

Do not underestimate the importance of cyber-attack capabilities.
I do not know how to defend a system if you are unaware of how to attack it.

[Filiol 2013b]

1.1 Context

Actors and Threats The Internet is a connected network of billions of devices. For the simplicity of administrating them, we plugged into this network of networks devices having an impact on the physical world: traffic control, power plants, gas stations, etc. Corporations and governments have data-stores connected to the Internet [Duwell 2013]. Banks and trading systems offer an interface with the Internet. If not secured, those systems make the Internet a playground for hackers with varying motivations (e.g., enemy governments, army, individuals paid by Mafia, etc.).

As security researchers, it is our duty to develop new techniques to protect better national assets such as: energy, money, communication and information. More

specifically, in this cyber war, we want to protect computer assets from attacks exploiting *vulnerabilities* (flaws in the system). A way to achieve such goal is to detect vulnerabilities, and more precisely to detect them as soon as possible. If they are present in our systems, we need to patch them to prevent exploitation (defensive security). If they are present in enemy systems, attackers may want to exploit them to gain additional privileges (offensive security). The source code of applications may not be available (e.g., when testing integrated or remote components). In such cases, we need to rely on black-box testing techniques. This thesis focuses on detecting certain classes of vulnerabilities in a black-box test context.

Security and Vulnerabilities Figure 1.1 shows a dependability tree according to the [Avižienis *et al.* 2004] taxonomy. We mark in **bold** our focus for this thesis. We detect errors and failures that affect *availability* (readiness for correct service),

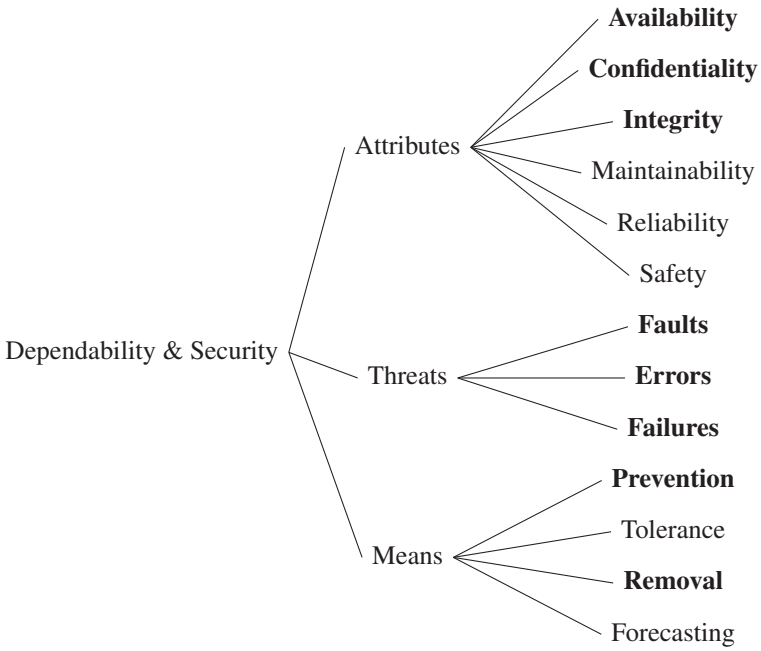


Figure 1.1: Dependability & Security Tree [Trivedi *et al.* 2009] and our **Focus**

integrity (absence of improper system alteration), *confidentiality* (absence of unauthorized disclosure of information). More specifically, we search for vulnerabilities in black-box test context. We address the automatic detection of Cross-Site Scripting (XSS).

1.2 Vulnerabilities

1.2.1 Panorama

A vulnerability is an application fault which ultimately will lead to a failure violating a security property that was supposed to always hold.

Afterwards we list a non-exhaustive panorama of vulnerabilities:

- **Code execution**: permit an attacker to force an application to execute code he created, because of a confusion between code and data. In such cases, the integrity of the executed code is violated. It generally means that the confidentiality and the integrity of the data processed by the application is also violated. It may also mean a violation of the availability property for the application. Code injection vulnerabilities include:
  - memory corruption (e.g., Buffer Overflow),
  - web command injection (e.g., Cross Site Scripting),
  - cross-format interpretation (e.g., GIFAR [Magazinius *et al.* 2013]).
- **Logical**: for example, in an authentication protocol, a parameter is lacking a security property (e.g., encryption, freshness, etc.), this makes the whole scheme vulnerable to a particular kind of attacks:
  - confidentiality: a credential is transmitted over an unencrypted channel, e.g., not over an SSL encrypted connection;
  - integrity: an unsigned authentication token permits a user to impersonate another one, e.g., `session_id`;
  - freshness: an action vulnerable to replay attack e.g., Cross-Site Request Forgery (CSRF)[Lin *et al.* 2009], [Armando *et al.* 2008].
- **User interface** : techniques trick the user to perform an action, whereas he believes performing another one, e.g., click-jacking [Rydstedt *et al.* 2010]. Such techniques violate the integrity of the user interface, thus of his actions.
- **Flawed access control**: depending on the path used to access an object, there may be a discrepancy in the way the access control is enforced. Such a situation frequently occurs when developers introduce new features in Web Browsers (e.g., [Heiderich 2012a, Heiderich *et al.* 2010, Heiderich 2013b]). Such vulnerabilities generally result in the violation of the integrity of objects. In the case of a Same-Origin Policy Bypass [Huang *et al.* 2010], the confidentiality of documents belonging to other security domains is also violated.
- **Flawed or weak cryptography**: e.g., sensitive information is sent in cleartext [Soltani 2013] or the implementation of a cryptographic algorithm is flawed [Thomas 2013]. Either the integrity, or the confidentiality are generally targeted.

In this thesis, we will search for code execution vulnerabilities.

1.2.2 About Code Execution Vulnerabilities

Code execution vulnerabilities arise because of the way an application is processing data, it may interpret part of it as code. Examples of code execution vulnerabilities include *Buffer Overflow* – a *memory corruption* vulnerability – and *Cross Site Scripting* (XSS) – a *Web Command Injection* vulnerability.

A former black-hat[1] used to make money with carding[2] and botnets[3]. He used to rely on XSS 55% of the time to take control of a website [Hansen 2013]. In order to gain access to a web application, he used XSS more frequently than memory corruption vulnerabilities. In terms of frequency, SQL injection was the third most common vulnerability category he used. All those three kinds of low-level vulnerabilities permit attacker controlled code execution. XSS and SQL injection belong to the family of Web Command Injection. SQL injection ranks first in the OWASP top 10 of 2013, and XSS ranks third [OWASP 2013b]. Whereas years ago, the focus was on desktop web application development, there is an increase in customized versions of web applications for mobile devices (e.g., Android, IOS applications). In this domain, developers seem to make similar mistakes. As a result, numerous mobile applications are sensitive to Web Command Injection vulnerabilities [Moulu 2013].

1.2.3 Web Command Injection

Web Command Injection (WCI) belongs to the code execution vulnerabilities. WCI is a family of vulnerabilities that affects applications interpreting a scripting language (e.g., HTML, SQL, Shell, PHP etc.) interpreters. WCI vulnerabilities are characterized by the possibility to escape a *confinement* within a grammatical structure. Example of Web Command Injection include Cross Site Scripting (XSS), SQL injection, PHP Code injection, etc.

Cross-Site Scripting (XSS) is one of the currently most dangerous web based attacks: it ranks third in the [OWASP 2013b] Top 10 vulnerabilities. [Zalewski 2011b] describes them as "one of the most significant and pervasive threats to the security of web applications." Criminals use XSS to spam social networks, spread malwares and steal money [Luo *et al.* 2009]. In 2013, XSS were found in Paypal, Facebook, and eBay [Kugler 2013, Nirgoldshlager 2013] [ZentrixPlus 2013]. We shall present XSS in Chapter 2.

[1] Unauthorized hacker having malicious intentions; of course, the notion of maliciousness is dependent on the entity which assesses it.

[2] The process of cloning credit cards.

[3] Network of bots, computer nodes controlled by an attacker.

Consequences of XSS an XSS vulnerability is activated by a maliciously crafted HTTP request, or a maliciously crafted Ajax request. Its exploitation provides to an attacker the capability of injecting arbitrary HTML code within portions of the web application. Thus, the victim web interpreter will execute a sub-interpreter code, which is controlled by the attacker. This permits:

- exfiltrating data (e.g., emails [Krebs 2012], authentication tokens [Naraine 2010], bank account password, contacts [Acunetix 2010], etc.);
- using the victim computer as a proxy or a node of a malicious network (e.g., spam relay, DDoS, exploiting websites, malware propagation [Faghani & Saidi 2009], mining bitcoin for the attacker, etc.);
- de-anonymizing a target: a browser can be uniquely tracked – up to a certain precision – via HTTP headers, available plugins and version, subset of interpreted codes [Nikiforakis & Vigna 2013] [Abgrall *et al.* 2012];
- exploiting a memory corruption vulnerability in a browser sub-interpreter to execute attacker controlled assembly code (e.g., [CVE-2008-1380 2008, CVE-2006-4565 2006] target JavaScript interpreters in browser and email client), to gain additional privileges (e.g., escape the browser process, obtain additional security tokens, etc.).

1.3 Objectives

Our main problem is:

How to improve the efficiency and precision of black-box security testing for automatically detecting XSS?

In order to address it, we face several sub-problems:

- sources: *on which parts of the inputs* to act for an efficient security testing?
- input sequences: *how to drive* the system into a desired state?
- maliciousness: *how to create* parts of inputs likely to exhibit a failure?
- test verdict: *how* and *where* to observe the effect of an input?
- confidence: *which criteria* characterize a precision in security testing?

1.4 Contributions

Our contributions are:

- an inference and slicing approach of particular control and taint flows for guiding XSS search;

- a combination of model inference and fuzzing for detecting Type-1 (reflected) & Type-2 (stored) XSS;
- an implementation of these approaches and their evaluation,
- which led to the discovery and responsible disclosure[4] of previously unknown vulnerabilities: four stored XSS (CVE-2013-7297[Duchène 2013a], [Duchène 2014d], [Duchène 2014c]), and of forty reflected XSS (CVE-2014-1599 [Duchène 2014a], [Duchène 2014c]), some of them impacting millions of users.

1.5 Dissertation Structure

We introduce the addressed problems in Chapter 2. We present the high level architecture of our approach in Chapter 3. Our approach for black-box XSS detection combines a particular control+taint flow inference (Chapter 4) and evolutionary fuzzing (Chapter 5). We evaluate this approach in Chapter 6. We provide an overview of related techniques in Chapter 7. We conclude and provide directions for future work in Chapter 8.

[4] In the responsible disclosure vulnerability model, vendors having a vulnerability in their product are allowed a period of grace before the security researcher who discovered this vulnerability makes it public.

CHAPTER 2

Problem Statement

The only acronyms that matter: RCE (Remote Code Execution), LPE (Local Privilege Escalation), COE (Continuation of Execution)

[Grugq 2013]

The importance of XSS may overstep that of memory corruption vulnerabilities.

[Heiderich 2013a]

Due to their wide existence and their high impact when exploited (an attacker is able to remotely execute arbitrary code in the victim's interpreter), we focus in this thesis on a particular case of web command injection: XSS. We define XSS in Section 2.1, and the problem of automatically detecting XSS in black-box in Section 2.5.

Web Command Injection (WCI) is a family of vulnerabilities that affects applications interpreting languages at run-time (e.g., HTML, SQL, Shell, PHP etc.). Since they compile and execute instructions at run-time, those are referred to as *interpreters*. Command injection vulnerabilities are characterized by the possibility to escape a *confinement* within a structure.

Each WCI subfamily has a name dependent on the context (i.e., the output grammar: database interpreter, browser interpreter, shell...). e.g., for the HTML grammar, WCI is named XSS.

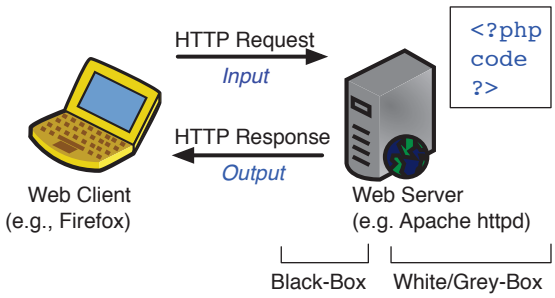


Figure 2.1: Black-Box Web Command Injection Detection

2.1 Cross Site Scripting (XSS)

The detection of XSS involves a *taint-flow* analysis on a *control-flow* graph.

Example A: an XSS involves a control-flow P0wnMe is a voluntarily vulnerable web application containing several XSS. Once authenticated, a user Peach can save a new message, view the saved ones, or logout. We illustrate several functionalities of the P0wnMe application in Figure 2.2, Figure 2.3 and Figure 2.4.

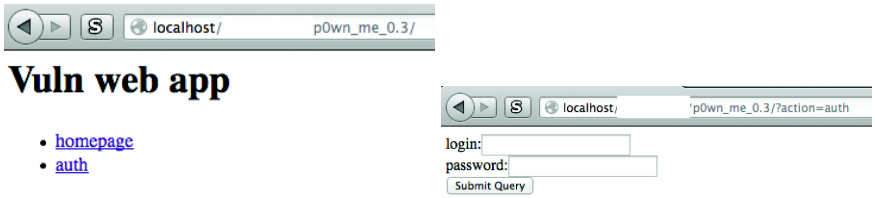


Figure 2.2: Screenshots of P0wnMe v0.3: login

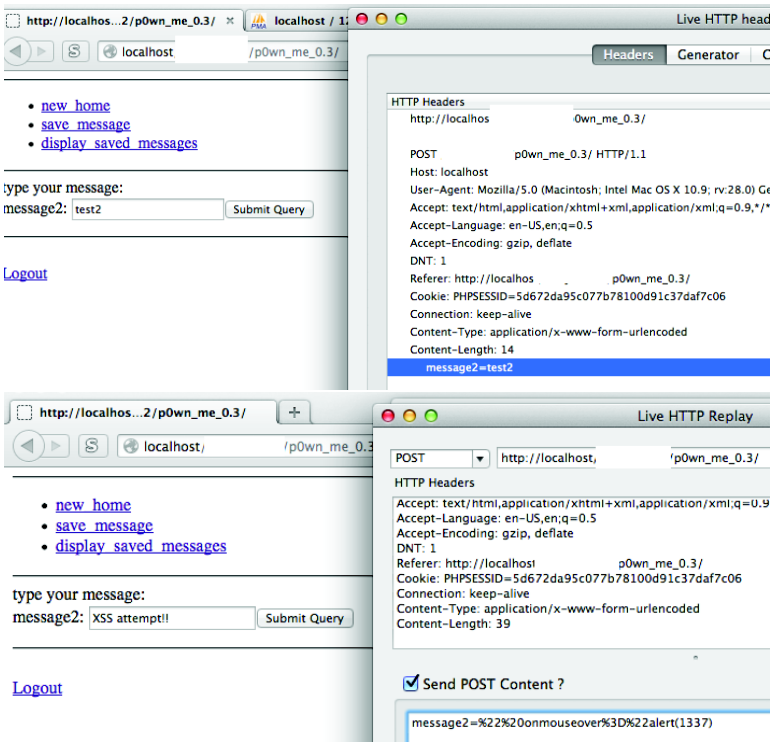


Figure 2.3: Screenshots of P0wnMe v0.3: Filtered Type-1 (reflected) Taint Flow

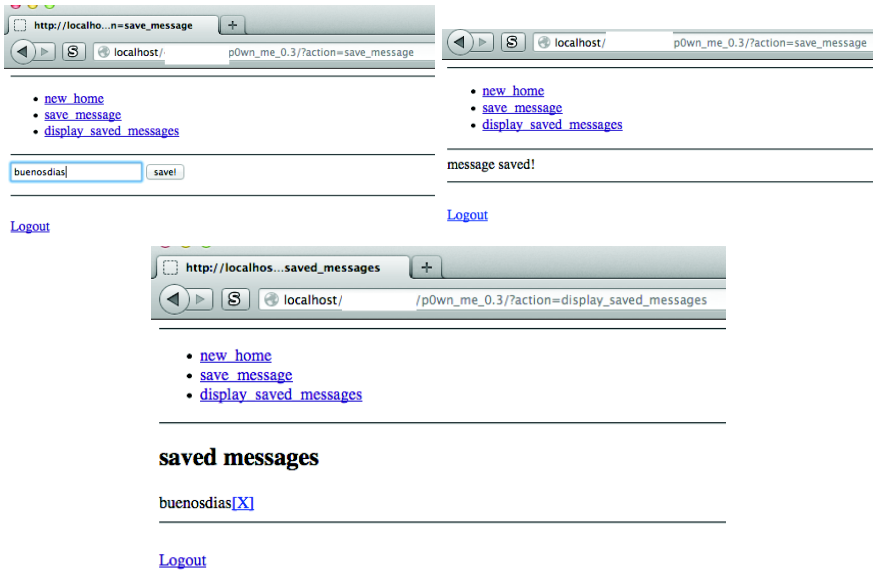


Figure 2.4: Screenshots of P0wnMe v0.3: Type-2 Taint Flow

Peach saves a note, e.g., buenosdias, by filling and submitting the form, i.e., sending the abstract input `POST /?action=save_message&msg=buenosdias` [1] (transition 7 → 17 in the control flow model shown in Figure 2.5) to the application. We describe in Chapter 4 how to construct such control flow models, where nodes represent pages and transitions HTTP requests. Such Control Flow Models (CFM) are different of assembly control flow graphs that most security engineers are used to work with, but CFM capture similar information at a higher level of abstraction. Later on, she lists the saved notes, by sending `GET /?action=get_messages` (transition 18 → 21). An extract of corresponding output is shown in Listing 2.1.

```
1   <H2>list of saved messages</H2>
2     buenosdias <A href="./?action=delete&id=1">[X]</A>
```

Listing 2.1: Excerpt of P0wnMe Output for the Transition 18 → 21

The value of the input parameter `msg`, sent in the transition $t_{src} = 7 \rightarrow 17$, is *reflected* in 18 → 21: we observe it into the output. This *reflection* is not filtered: the exact value sent in 7 → 17 is copied into the output of $t_{dst} = 18 \rightarrow 21$.

[1] We highlight text to indicate that it is part of a taint flow (partial string copy).

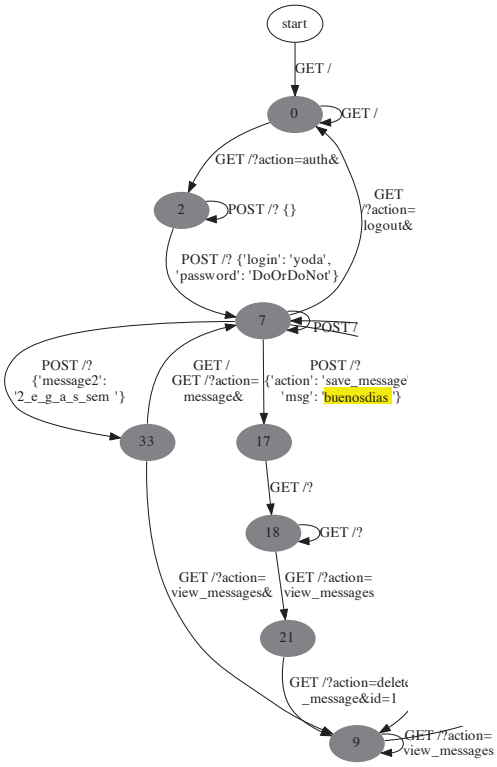


Figure 2.5: Extract of a Control Flow Model of the P0wnMe Web Application

Moreover, in this application, notes are shared between users. Thus, an attacker Koopa Troopa would attempt to send a malicious `msg` value to escape the *confinement* (in Listing 2.1, a reflection is constrained in a specific context: outside tags, before the `<A>` tag). An example of malicious input is $t_{src} = 7 \rightarrow 17$ (`POST /?action=save_message&msg=buenosdias <script> alert(1337)</script>`).
An excerpt of the corresponding output for the subsequent transition $t_{dst} = 18 \rightarrow 21$ is `...of saved messages</h2>buenosdias <script>alert(1337)</script><a href="./?action=delete....`
When Peach's browser (the victim) parses this output, it executes the *code* introduced by the attacker, and a messagebox is displayed, as shown in Figure 2.6.

In order to detect XSS, we need to navigate in the web application. Thus, we need information about the *control flow* of the application. The problem is that most deployed web applications lack formal documentation: a formal behavioral model is rarely available. However, such models improve the ability to perform a pertinent security testing campaign for an application [Takanen *et al.* 2008], espe-

Figure 2.6: Successful exploitation of an XSS in the P0wnMe application

cially if they combine control and taint information [Bekrar 2013b]. Thus the first sub-problem to address is:

XSS.1 Navigating in the Application:
→ How to obtain a model of the application? What kind of models are appropriate for detecting XSS?

Example B: an XSS involves a taint flow A fictive website `http://yoshi.jp` suffers from an XSS vulnerability. A necessary condition for an XSS is a reflection, i.e., a taint-flow i.e., a partial input copy from an attacker controlled input parameter to a sink, part of an output of the web application, as illustrated in Figure 2.7. Our notion of taint flow is different to the traditional taint propagation rules for each assembly instruction in grey-box contexts [Newsome & Song 2005, Xu *et al.* 2006]. We define our taint notion in web applications in black-box more precisely in Definition 3.

Client → Server:

```
1  ...
2  GET /nice.html?name=birdo HTTP/1.1
3  Host: yoshi.jp
```

Client ← Server:

```
1  ...
2  Hello birdo!
```

Figure 2.7: The Input Parameter name is reflected into the output of the web application

Koopa Troopa wants the victim Peach to execute a code that he controls. He observes that the reflection is located in a structure “text node, outside a tag”.

He wants to "escape" this text node, and includes additional interpreter nodes such as JavaScript (JS) ones. Thus he crafts the parameter `name` with the value `lakitu<script>alert('evil');</script>`. To achieve exploitation, he creates a page hosted on `koopatroopa.fr`, which will force the user browser to perform this malicious request to the vulnerable website (hence the cross-domain), once the code of `koopatroppa.fr` is interpreted (see Figure 2.8).

Client → Attacker:

```
1 GET / HTTP/1.1
2 Host: koopatroopa.fr
```

Client ← Attacker:

```
1         ...
2 <iframe src="http://yoshi.jp/nice.html?name=
      lakitu<script>alert('evil!');</script>">
3         ...
```

Client → Server:

```
1 GET /nice.html?name=lakitu<script>alert('evil!');</script> HTTP
      /1.1
2 Host: yoshi.jp
```

Client ← Server:

```
1 ...
2 Hello lakitu<script>alert('evil!');</script>!
```

Figure 2.8: Successful exploit of an XSS

Thus, in order to detect XSS, we need to address the sub-problem:

XSS.2 Achieving a Test Verdict
→ How to infer the taint?
→*XSS.2.1* ***Where*** *are the potential sinks (reflections)?*

2.2 Definitions

We abstract an actual Web site as a *Web Application*. The Web Application receives a *spiderlink* and replies with a *page model*. As illustrated in Figure 2.9, spiderlinks represent the HTTP requests sent to the concrete Web site, and page models abstract the HTTP responses.

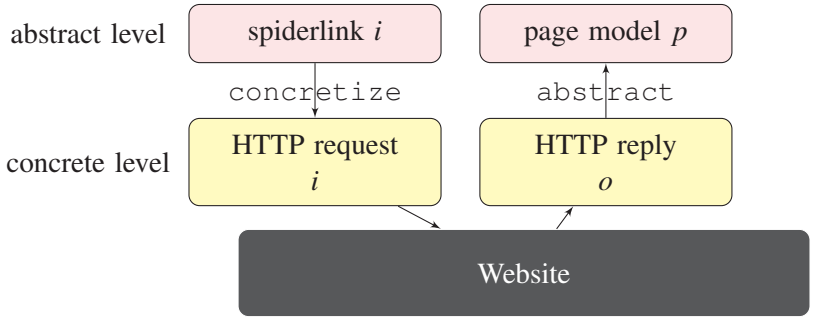


Figure 2.9: Abstraction and Concretization Functions for Web Applications

2.2.1 Spiderlink and Page Model

We define `abstract(o)` which abstracts an HTTP reply o into a *page model* p. As illustrated in Figure 2.10, a page model is a prefix tree containing several spiderlinks (i.e., abstract links from `<A>` tags or abstract forms from `<form>` tags). We also define `concretize(i)` which produces an HTTP request *req* from a *spiderlink* i.

Input, `concretize`(Spiderlink i) Let Σ be an alphabet. Each spiderlink is built from a link or a form.

Definition 1 Spiderlink

A *spiderlink* is a couple composed of:

- one *action*: the substring before the `?` of an Hypertext Transfer Protocol (HTTP) Uniform Resource Locator (URL)
- a list of input parameters
  $(name, value, method) \in \{\Sigma^*\}^2 \times \{$`GET, POST, COOKIE, HEADER`$\}$

Output, abstraction to Page Model p A page model is an abstraction of a concrete output of a web application. It contains spiderlinks.

Definition 2 Page Model

Let $o \in O$ be an output of a web application. Let $F(o)$ be the set of links and forms in o. A page model $p = abstract(o)$ is a prefix tree which is built from the set of spiderlinks and dompaths obtained from $F(o)$. It has at least six levels including the root node.

For each spiderlink $f \in F(o)$, a set of nodes is added to the tree. Each set consist of the following nodes or groups of nodes, ordered from the immediate children of the root node to the deepest ones:

- *dompath* is a node, child of the root node. Its value is a string $\in \{/([a-z] \cup \{/\})^*\}$, the shortest path in the Document Object Model (DOM) from the root to the `<A>` or `<FORM>` tag.
- *action* is a tree of nodes, of depth ≥ 1. Its root is a child of a dompath node. Each node of an action subtree has a string value contains a part of an HTTP URL before the `?` split by **/**
- *params* is a node, child of an action node. Its value is a list of strings: the list of parameter names.
- *values* is a node, child of a params node. Its value is a list of strings: the list of parameter values.
- *methods* is a node, child of a values node. Its value is a list of parameter methods, each element $\in$ {`GET, POST, COOKIE, HEADER`}

Consider the HTML ouput in Listing 2.2. The corresponding page model is shown in Figure 2.10. The left side shows the browser rendering, while the right side represents the page model. It contains four spiderlinks. The prefix tree representation makes easier the identification of similarities between spiderlinks and between page models.

```
1  ...
2  <html>
3  ...
4  <body>
5
6  <div class='menu'>
7    <span id='menu-left'>
8      <a href='/'>Home</a>
9
10   </span>
11   <span id='menu-right'>
12       <a href='/login'>Sign in</a>
13       | <a href='/newaccount.gtl'>Sign up</a>
14   </span>
15 </div>
16
```

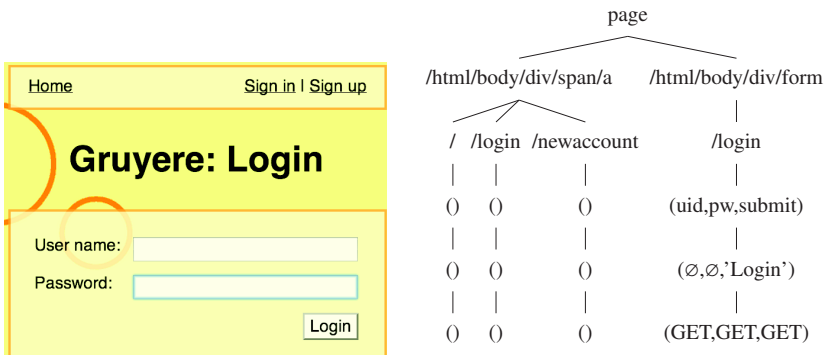


Figure 2.10: Abstraction: graphical representation of the output and corresponding Page Model.

```
<div>
<h2>Gruyere: Login</h2>
</div>

<div class='content'>
<form method='get' action='/login'>
<table><tr><td>
    User name:
  </td><td>
    <input type='text' name='uid'>
</td></tr><tr><td>
    Password:
  </td><td>
    <input type='password' name='pw'>
</td></tr><tr><td></td><td align='right'>
    <input type='submit' name='submit' value='Login'>
</td></tr></table>
</form>
</div>
</body>
</html>
```

Listing 2.2: The output o (extract)

2.2.2 Taint

The taint is a metadata information between inputs and data handled by the application [Livshits 2012]. It designates the possibility of an input to explicitly influence such data. The taint is a dynamic notion which flows between data. It is generally a dynamic white-box (assume the availability of the application source code) or grey-box (assume the availability of the application code) notion, as in [Rawat & Mounier 2010, Bekrar *et al.* 2012], although in our case we will perform taint inference dynamically in black-box (see Section 4.3).

When considering taint propagation in grey-box or white-box, three sub-problems arise:

- **Taint Sources**: When is an object *directly influenced* by inputs? In Table 2.1 we list examples of taint sources.
- **Taint Propagation rules**: When is an object *indirectly influenced* by inputs?
- **Taint Removal**: When is an object *not influenced* by inputs?

Taint tracking is widely used in white-box/grey-box test context for vulnerability runtime detection, e.g., for XSS [Vogt *et al.* 2007] and for Memory Corruption Vulnerabilities [Bosman *et al.* 2011].

Practical data tainting on important sized applications may only consider explicit value influence (e.g., assignments) in taint propagation rules, and not considering indirect value influence (e.g., resulting from a conditional check). Indeed, such taint tracking systems aim at avoiding too numerous objects to be tainted, in order to reduce the number of false positives in the test verdicts [Haller *et al.* 2013].

In black-box test context, as we cannot track the taint flow from a source to a sink, we have to infer it, if possible. In Section 4.3, we provide information on taint inference for XSS.

In Table 2.1, we list examples of code execution vulnerabilities, their taint source, the code at server side, and sometimes how to infer knowledge in black-box.

Vulnerability	**Taint Source**	**Sink & Tainted Argument (white-box)**	**Inference (black-box)**
Memory Corruption	File	`read(han, buf, nb)`	
	Network read	`recv(sock, buff, len, flag)`	
	Keyboard input	`scanf("%d", &num)`	
XSS	GET/POST parameters values	`print( $_POST['email'] )`	input copied to the output
SQL injection	Cookie parameter values	`sql_query( $_COOKIE['sess_id'] )`	error message
Shell Injection	Parameter values	`shell_exec( $_POST['action'])`	error message

Table 2.1: Examples of Vulnerabilities and Taint Markers

In black-box test context, XSS vulnerabilities arise when a tainted *data* (resulting of a partial copy of an input parameter value, a string of characters) appears in at least one substring of one output (HTML code), and gets executed by the browser as *code*.

Definition 3 Taint

Let x_{src} and o_{dst} be two strings. $x_{src} \rightsquigarrow o_{dst}$ denotes x_{src} taints o_{dst}. Our notion of taint measures the similarity between two strings. It uses string distance functions.

We define more precisely our notions of taint in Definition 5 and Section 5.2.

2.2.3 Vulnerability and Exploit

A vulnerability is a fault leading to an error. If a fault is traversed by a taint flow, which is used to stimulate the fault, this will lead to a failure. Thus the problem of searching vulnerabilities can be addressed by searching for sinks, and then searching for inputs activating those sinks in a way prone to exhibit failures.

We are interested in vulnerabilities which violate the *code integrity property*. Such vulnerabilities arise due to confusion between data and code: the interpreter will consider attacker controlled data as code and thus execute it.

Cross Site Scripting (XSS) is a Web Command Injection vulnerability within the HTML grammar. An XSS permits attacker controlled code execution at client side. An example of sink for XSS is the PHP `print()` function:
`<?php print( $_GET['message'] ); ?>`.
For some malicious messages, once the client browser renders the webpage, the property stating that the content of the message variable should not be executed by the browser is violated.

An *exploit* is an input activating a vulnerability s.t. the application will violate a security property (in our case the integrity of the code executed at browser side, as it will execute a *payload* resulting of the input).

Within a given class, some vulnerabilities are more *complex* to find than others. The complexity of a vulnerability is an increasing metric w.r.t. the minimum number of traversed states of the control flow model to violate a security property. When searching for Web Command Injection, the filter (sanitizer) and the number of distinct traversed nodes in the control flow models are factors influencing the complexity of a vulnerability. When searching for Memory Corruption, the number of traversed jump instructions affects the difficulty in finding such vulnerabilities.

2.2.4 Web Application, Reflection, Syntactic Confinement, XSS

Figure 2.7 illustrates a *Reflection* (i.e., a taint flow from an input parameter value to an output). A reflection can be tracked (e.g., in white-box test context), or inferred (e.g., in black-box test context). We formally define a reflection in Definition 5.

Definition 4 Web Application (WA)

Let P be a set of page models, and I be a set of spiderlinks.
A Web Application (WA) $W = (N, n_0, T, I, P, \Pi)$, is a graph:

- N is a set of nodes. $n_0 \in N$ is the initial node of the application.
- Each transition $t = (n_a, i, n_b) \in T \subset (N \times I \times N)$.
- $\Pi : N \rightarrow P$ is a mapping which to each node n associates a page model p.

In Section 2.5.2, we elaborate on the reasons why we consider that a reflection only involves one taint source parameter. In Chapter 8, we provide insights on how to extend our work in the case of several taint sources.

Definition 5 Reflection

Let Σ be an alphabet. Let $W = (N, n_0, T, I, P, \Pi)$ be a Web Application (Definition 4). Let $S_I = [i_0, .., i_{src}(x_{src}), .., i_{dst}]$ be a sequence of spiderlinks ($\in I^+$) and $S_O = [o_0, .., o_{src}, .., o_{dst}]$ ($\in \Sigma^{++}$) the corresponding sequence of concrete outputs (trace). When submitting the input i_{dst}, the obtained concrete output is $o_{dst} \in \Sigma^+$. Let $\delta \in \mathbb{N}$ be a threshold. Let x_{src} be an input parameter value received in the input i_{src} of the source transition t_{src}. The execution of S_I on W terminates with $t_{dst} = (n^s_{dst}, i_{dst}, n^e_{dst})$.
$(x_{src}, t_{src}, t_{dst}, o_{dst})$ is a δ-reflection if:

- x_{src} taints o_{dst}, there is a partial copy of length $\geq \delta$ of x_{src} into o_{dst} i.e., $\exists y \in \Sigma^+$
  - $|y| \geq \delta$
  - $(y \sqsubseteq x_{src}) \wedge (y \sqsubseteq o_{dst})$, where $x_1 \sqsubseteq x_2 \iff \exists x_3, x_4 \in \Sigma^{*^2}, s.t.$ $x_2 = x_3 x_1 x_4$
- $\Pi(n^e_{dst}) = abstract(o_{dst})$

We denote it as: $(x_{src}, t_{src}) \rightsquigarrow_\delta (o_{dst}, t_{dst})$.

A sanitizer/filter is a mechanism at server-side which validates and eventually modifies (mutates) a fuzzed value before it is reflected. Sanitizers may modify the input parameter values, e.g., by removing some characters having a special meaning in the considered grammar (e.g., (‘,”,> ,<) in the HTML grammar for XSS vulnerabilities). A common mistake when building such filters is to overlook the *context* of the reflection [Weinberger *et al.* 2011a] (for instance always applying a given string transformation regardless of where the reflection happens a.k.a. “Context-Insensitive Auto-Sanitization”. This may result in a false sense of security, in which the

developer believes to be protected from XSS, whereas the sanitizer is flawed [Weinberger *et al.* 2011a]. We illustrate an example of flawed sanitizer in Listing 2.3.

```
1 <?php function webapp_filter($str) {
2     if(eregi('"|`|>|<|;|/',$str)) {
3         $filtered_str = "XSS attempt!";
4     } else {
5         $filtered_str = str_replace(" ","",$str);
6     }
7     return $filtered_str;
8 } ?>
```

Listing 2.3: A vulnerable sanitizer in P0wnMe

Since filter/sanitizer may change significantly the fuzzed value before it gets reflected, we need to address the sub-problem:

XSS.2 Achieving a Test Verdict
*→XSS.2.2 Can we **exploit a potential sink?** (i.e., find an input which bypasses the filter) How to infer the taint in the presence of filter?*

Generating an input which is able to partially bypass the filter by partially being copied into the output is not a sufficient condition for a successful XSS exploit. Indeed, the reflected value has to "escape" the structure in which it was *confined*. We name this objective *non syntactic confinement*.

Figure 2.11 illustrates two reflections. The first one is syntactically confined according to the grammar (see the production rules in Figure 2.12), and the second one is not. Graphically, the first reflection is syntactically confined because there exists one non-terminal (**TEXT**) s.t. the whole produced sub-tree is tainted. This is not the case for the second reflection: the first common parent non-terminal of **TEXT** and **SCRIPT** is **START**, and its sub-tree is not fully tainted.

[Su & Wassermann 2006, Wassermann 2008] formalized the problem of web command injection with the notion of *Syntactic Confinement* (Definition 6). We use the following notations: Let $G = (V, \Omega, S, R)$ be a context-free grammar with nonterminals V, terminals Ω, a start symbol S, and productions R. Let "$\Rightarrow_G$" denote "derives in one step" s.t. $\alpha A\beta \Rightarrow_G \alpha\gamma\beta$ if $A \rightarrow \gamma \in R$, and let "$\Rightarrow_{*G}$" denote the reflexive transitive closure of "$\Rightarrow_G$". If $S \Rightarrow_{*G} \gamma$, then γ is a "sentential form".

Definition 6 Syntactic Confinement

Given an unambiguous grammar $G = (V, \Omega, S, R)$ (V non terminals, Ω is an alphabet of terminals, S a start symbol, R production rules), a word $\omega = \omega_1\omega_2\omega_3 \in \Omega^*$, ω_2 is syntactically confined in ω iff there exists a sentential form $\omega_1 X \omega_3$ such that $X \in (V \cup \Omega)$ and $S \Rightarrow_{*G} \omega_1 X \omega_3 \Rightarrow_{*G} \omega_1\omega_2\omega_3$.

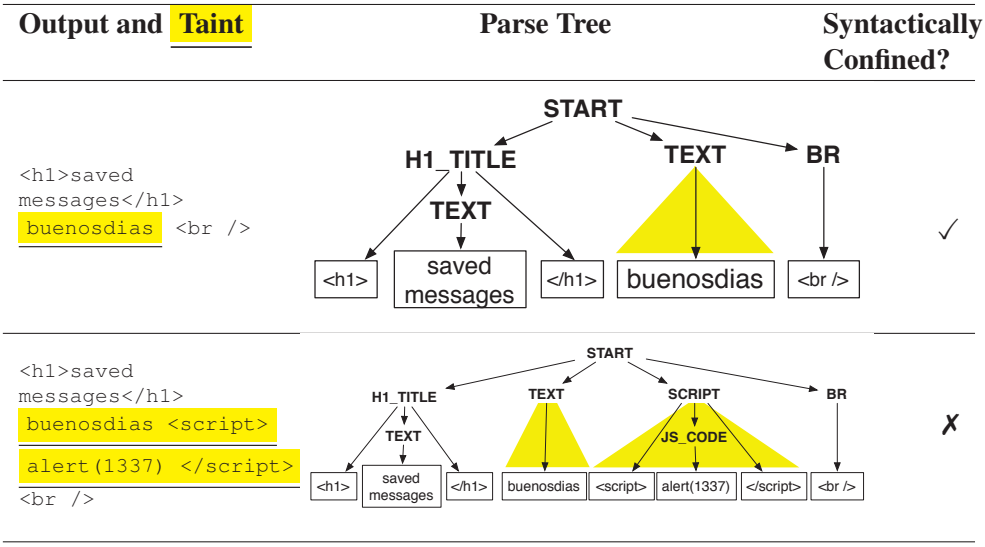


Figure 2.11: Syntactic Confinement of two Reflections in G =HTML

START → [0:20](H1_TITLE | TEXT | BR | SCRIPT)
H1_TITLE → "<h1>" TEXT "</h1>"
TEXT → "a" | "b" | ...
BR → "
"
SCRIPT → "<script>" JS_CODE "</script>"

Figure 2.12: Extract of the HTML Grammar Production Rules

In Definition 7, we give a restrictive definition of the notion of Web Command Injection (WCI).

Definition 7 Web Command Injection (WCI)

Let M be a Web Application (Definition 4), $R_{efl}(M)$ be a set of reflections (Definition 5) in M, and G be a grammar. Let $\delta \in \mathbb{N}$ be a threshold. Let $r = (x_{src}, t_{src}, t_{dst}, o_{dst}) \in R_{efl}(M)$ be a δ-reflection (Definition 5), where for a given trace, x_{src} is a concrete input parameter value of the transition t_{src}, and o_{dst} is the concrete output value of the transition t_{dst}. Let $Z = taintedsub(x_{src}, o_{dst})$ be the set of substrings of length $\geq \delta$ in o_{dst} which are tainted by x_{src}.
r is a Web Command Injection (WCI) w.r.t. to G, if $\exists\, z \in Z$, s.t. z is not syntactically confined in o_{dst} w.r.t. G.

A WCI permits to violate the code integrity property at the level of G.

Type-1 (reflected) and Type-2 (stored) XSS exist since applications handle dynamic data (i.e., since the first cgi-bin scripts appeared in web applications). As of today, this problem is still unsolved: no scanner detects 100% of the XSS in all

Definition 8 Cross Site Scripting (XSS) Vulnerability

We define a Cross-Site Scripting (XSS) as a Web Command Injection (Definition 7) in which the output grammar G is HTML (and the interpreted grammars: e.g., JavaScript, CSS, etc.).

Definition 9 XSS Types

XSS Type	Transition (request)	Characteristic
"Pure" Type-0 / DOM XSS	Ajax	In a reflected DOM-XSS, the taint flow does not involve an HTTP request. Nodes are DOM states and transitions are Javascript function or event calls.
Type-1 / Reflected XSS	HTTP	The source and destination transitions are the same: $t_{src} = t_{dst}$. At least one HTTP request is involved.
Type-2 / Stored XSS	HTTP	The source and destination transition are different: $t_{src} \neq t_{dst}$. At least two HTTP requests are involved.
Stored DOM-XSS	Ajax + HTTP	At least one Ajax request and one HTTP request are involved.

web applications. Type-0/DOM XSS exist since web application execute dynamic code at client side (e.g., JavaScript Ajax transitions, such as Facebook). DOM XSS involve Ajax transitions. **In this thesis, we only focus on Type-1 (reflected) XSS and Type-2 (stored) XSS.**

[Heiderich *et al.* 2013] browser parser quirks induces transformations conform to the definition of the categories mentioned in Definition 9.

2.3 Fuzzing

Fuzzing is the automatic generation and evaluation of abnormal inputs in order to trigger the targeted vulnerability family(ies). Fuzzing is sometimes named as "act of software torture" [Vuagnoux 2005]. The term was coined by Barton Miller [Barton *et al.* 1989, Forrester & Miller 2000].

When searching for XSS in black-box, we will create fuzzed inputs from control and taint flows knowledge in order to escape the syntactic confinement of reflections. Thus we need to answer the following sub-questions:

XSS.3 Creating Fuzzed Inputs:

- *XSS.3.1* ***Where*** *to fuzz inputs?* Which inputs to select? On which parts of those inputs to act?
- *XSS.3.2* ***How*** *to fuzz inputs?* How to act on specific parts of those inputs?
- *XSS.3.3 How to* ***prioritize*** *inputs fuzzing?* Which potential sinks should we test first?

2.4 Other Web Command Injection Vulnerabilities

XSS is one vulnerability in the Web Command Injection family. We list other sub-categories in Table 2.2, such as SQL injection, Shell Command Injection, PHP Code Injection etc.We believe that the work applicable to XSS can also be applied to those types of WCI.

Vulnerability	Grammar	Sink	Similar Vuln. in
Cross Site Scripting (XSS)	HTML (& sub-grammars)	print, echo	
SQL Injection (SQL_i)	SQL	sql_query, etc.	LDAP, NoSQL, etc.
Shell Command Injection	bash, sh, zsh etc.	exec	
XML External Entity (XXE)	XML	XML processor	
PHP Code Injection	PHP	eval	Ruby, Python, etc.

Table 2.2: Sub-Categories of the Web Command Injection Vulnerability Family

2.5 Summary of Addressed Problems

Automatically detecting XSS is an open problem. In the case of access to the source code, white-box techniques range from static analysis to dynamic monitoring of instrumented code. If the code or the binary are inaccessible, black-box approaches generate inputs and observe responses. Such approaches are independent of the language used to create the application, and permit a generic harness setup. As they mimic the behaviors of external attackers, they are useful for offensive security purposes, and may test defenses such as web application firewalls.

Automated black-box security testing tools for web applications have long been around. However, even in 2012, the fault detection capability of such tools is

low: the best ones only detect 40% of non-filtered Type-2 XSS, and 1/3 do not detect any [Bau *et al.* 2010, Bau *et al.* 2012]. This is due to an imprecise learned knowledge [Doupé *et al.* 2012], imprecise test verdicts, and limited sets of attack values [Duchène *et al.* 2013b].

Thus there is a need for methods which detect Type-2 XSS (and server-side filtered Type-1 XSS) using a black-box test context.

Problem:
→ How to automatically detect Type-1 and Type-2 XSS in web applications, with a black-box test context?

2.5.1 Problems

According to the previous discussion, in order to effectively address XSS detection, the following sub-problems must be addressed:

- **XSS.1 Navigating in the Application**: In order to detect XSS in web application, we need to navigate in the application. Thus, we need information about the *control flow* of the application. The problem is that most deployed web applications lack formal documentation: a formal behavioral model, such as an FS (Definition 5), is rarely available. However, such models improve the ability to fuzz an application.
  → How to obtain a model of the application? What kind of models are appropriate for detecting XSS?

- **XSS.2 Achieving a Test Verdict**:

  Server-side sanitizers may perform significant string transformations between the fuzzed value and the reflection. Since such reflections are hard to observe, there is a risk of false negative in the taint inference, thus in the test verdict.
  → How to perform a test verdict in the case of filtered reflections?

  - *XSS.2.1* ***Where*** *are the potential sinks?*
  - *XSS.2.2 Can we* ***exploit a potential sink?*** *How to infer the taint in the presence of filter?*

- **XSS.3 Creating Fuzzed Inputs**: When creating fuzzed inputs, we need to answer the following sub-questions:

  - *XSS.3.1* ***Where*** *to fuzz inputs?* Which inputs to select? On which parts of those inputs to act?
  - *XSS.3.2* ***How*** *to fuzz inputs?* How to act on specific parts of those inputs?

- *XSS.3.3 How to **prioritize** inputs fuzzing?* Which potential sinks should we test first?

The core of this thesis is the automatic black-box detection of vulnerabilities that permit attacker controlled code execution. Those include cross-site scripting.

2.5.2 Hypotheses on the Web application

- **Reset**: as we want to replay some input sequences, we assume the ability to reset the application in its initial state. Candidate solutions include applying a virtual machine snapshot, but also killing the application, restoring the database in its initial state and starting it again.
- **Defensive Mechanisms**: Since we are interested in finding XSS vulnerabilities with a black-box test context, and since the deployment of many counter-measures is very low as of today (for more details, see Table 8.1 in Chapter 8), we assume that the only counter-measure which may be present in the tested web application is server side sanitizer. We believe that our approach for addressing this problem (see Chapter 4 and Chapter 5) could also address situations when a Web Application Firewall (WAF) is present.

CHAPTER 3

Our Proposal

> Nobody ever defended anything successfully, there is only attack, attack and attack some more.
>
> Gen. George S. Patton ; 1885–1945

> Glad to see more and more companies/researchers selling 0Ds to Govs. Software vendors are losing the game but they are not yet aware of that.
>
> [Bekrar 2013a]

> As you've probably noticed, I'm basically lazy which is why I like fuzzing.
>
> [Miller 2010]

The automatic detection of software vulnerabilities involve numerous combinatorial problems [Filiol 2013a]. This also holds for the automatic detection of XSS in a black-box test context. Due to these numerous problems and to the lack of formal knowledge, we propose domain-based engineering [Czarnecki *et al.* 2000] approaches which use heuristics guided by the practical experience of penetration testers. Such approaches may not be sound, but still are safe to be used in practice.

In this chapter, we briefly justify our reasoning for addressing the aforementioned sub-problems.

Our approach for automatically detecting XSS in a black-box test context consists of two steps: "crawling" infers the *control flow* and the *taint flows* of the application, then "fuzzing" generates malicious inputs to exhibit vulnerabilities.

3.1 Control and Taint Flow Model Inference

Our first step constructs a model of the web application. This is achieved by a combination of crawling and taint analysis. In order to do this, we need to address the challenges XSS.1, XSS.2.1, and XSS.3.1, among those expressed in Section 2.5.1 (page 33).

- **XSS.1 Navigating in the Application**: [Doupé *et al.* 2012] showed that black-box WCI security scanners perform poorly due to a lack of precise knowledge about the applications they are testing. How to learn knowledge for driving the application?

- **XSS.2 Achieving a Test Verdict**: XSS are characterized by the fact that once an input has been sent, its effect cannot always be observed right away (e.g., we may need to drive the application in another state). By answering to the question *XSS.2.1* ***Where*** *are the potential sinks?*, we can prioritize the locations where to invest more efforts in computing the test verdict.
- **XSS.3 Creating Fuzzed Inputs**: a naive fuzzing (e.g., mutating all input parameters) may spend too much testing resources when focusing on non-promising parts of the application. Therefore, we need to address the sub-problem: *XSS.3.1* ***Where*** *to fuzz inputs?* We also partly address the subproblem *XSS.3.3 How to* ***prioritize*** *inputs fuzzing?*

To answer these sub-problems, we propose a reverse engineering approach. Reverse Engineering is "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [Chikofsky *et al.* 1990]. Since we are in a black-box test context, reverse-engineering can be achieved by means of inference.

In order to address XSS.1, we propose an extension of [Doupé *et al.* 2012] for inferring the control flow of the application. Then, in order to address *XSS.3.1*, we propose to extend the previously obtained control flow model with a taint flow inference for indicating the reflections. The outcome is a hybrid control+taint flow model. Lastly, from this hybrid model, we generate input sequences fuzzing on a specific point and directing toward another point to observe, thus providing an answer to *XSS.2.1*. We summarize these choices in Figure 3.1, and develop it in Chapter 4.

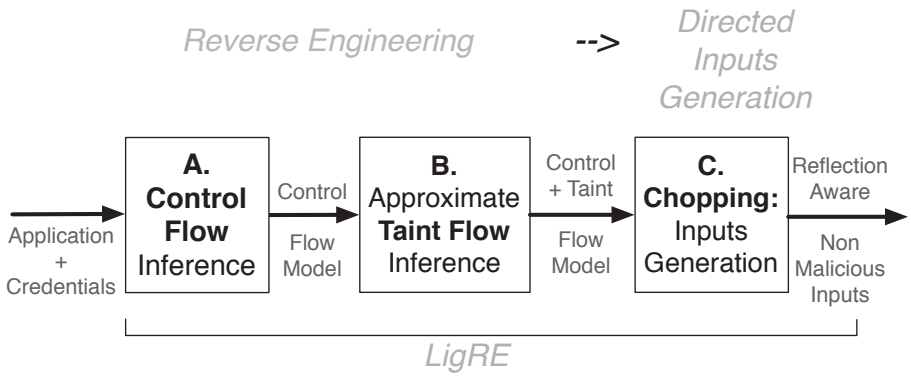


Figure 3.1: LigRE: Control+Taint Flow Model Inference

Example Most of considered open-source black box web scanners (Skipfish and Wapiti) fail at detecting the P0wnMe XSS presented in Section 2.1. The main reasons are imprecise application behavior awareness (some scanners do not navigate properly, and do not observe the reflections), imprecise test verdict (e.g., Skipfish

considers a page model change to be a sufficient condition for XSS), and limited set of fuzzed values (unaware of the output structure or the filters). Our approach overcomes the first limitation using a combination of control flow inference, taint flow inference and a guided fuzzing.

In step A in Figure 3.1, our tool called LigRE infers a Control Flow Model (Control Flow Model (CFM)) in the form of a colored automaton (nodes and continuous arrows of Figure 3.2, where nodes represent webpages/outputs and transitions represent inputs/HTTP requests), up to a tester defined precision. Then in step B, LigRE walks through the model by generating HTTP requests and submitting them to the application. The corresponding responses (HTTP replies) are recorded. Taint flows of sent input parameter values are inferred on the outputs, and annotated on the model (blue dashed lines on Figure 3.2).

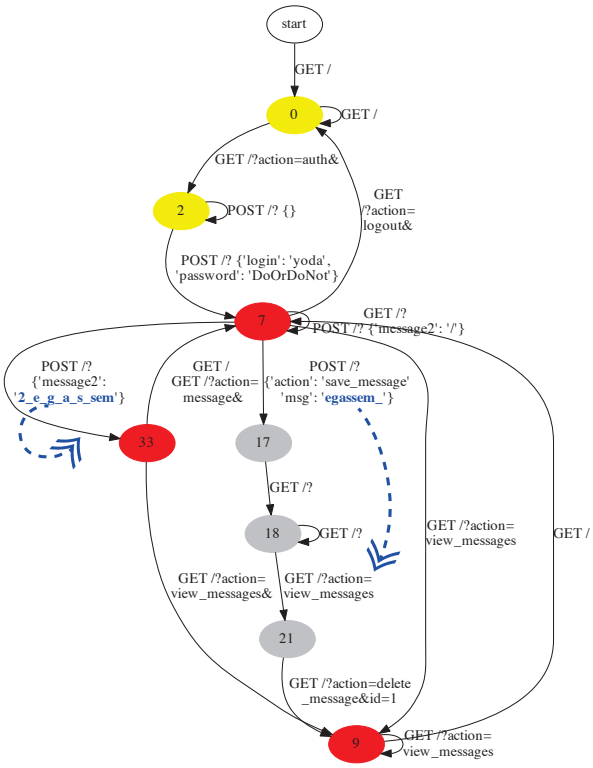


Figure 3.2: Extract of the CTFM for the P0wnMe application

In order to address *XSS.3.3 How to **prioritize** inputs fuzzing?*, the chopping step computes model slices (see Figure 3.3), and prioritizes them. Each slice is composed of a prefix and a suffix. For instance, the prefix $[0 \rightarrow 2, 2 \rightarrow 7]$ and the suffix $[7 \rightarrow 17, 17 \rightarrow 18, 18 \rightarrow 21]$. LigRE sends the prefix to the application, then pass the authentication credentials (e.g., cookie) to a fuzzer (e.g., w3af [Riancho 2011] or KameleonFuzz) and limits its scope to the suffix. Those slices permit to drive

the tested application toward the originating transition of an inferred reflection t_{src}, and to constrain the fuzzing towards the transition t_{dst} to observe the reflection.

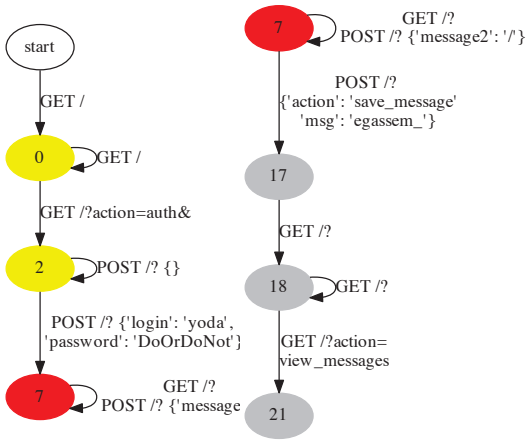


Figure 3.3: A chopping slice produced by LigRE, during the step C, for the P0wnMe application (prefix on the left part, and suffix on the right)

3.2 Evolutionary XSS Fuzzing

Once we have a model, we use it for generating fuzzed input sequences. In the current section, we illustrate this fuzzing process. We address the challenges XSS.3.2, XSS.3.3 and XSS.2 (Section 2.5.1, page 33).

- **XSS.3 Creating Fuzzed Inputs** For addressing the question *XSS.3.2* ***How*** *to fuzz inputs?*, we propose to reuse the hybrid control+taint flow model. Indeed such models notably provide information about the reflection context (e.g., the HTML structure of the reflection: outside a tag `<b>Hello Lakitu</b>`). Fuzzing exists with various flavors: random, anomaly operators, grammar-based. Since successful XSS exploits need to respect some HTML constraints, we choose to generate fuzzed values with an attack grammar. Thus the search space is composed of the reflections and the attack grammar.

  For addressing *XSS.3.3 How to* ***prioritize*** *inputs fuzzing?*, we can use metrics such as the rarity of a reflection, and the "injection power" of a reflection (e.g., how many different *grammar meaningful* HTML constructs are reflected?). We integrate such metrics in the fitness function of a genetic algorithm which captures characteristics of the best inputs and evolves them.

- **XSS.2 Achieving a Test Verdict** The input sequence includes point(s) where to observe the effect of a fuzzed input. Those are outputs of the web application. Several Black-box scanners, e.g., [Zalewski & Heinen 2009, Riancho 2011], only search for a verbatim reflection (exact string recopy) in the HTML code present in the body of an HTTP Reply. As this does not provide enough information about the ability to execute attacker controlled code, these scanners are likely to obtain fuzzy test verdicts. Indeed, server-side sanitizers may significantly transform fuzzed inputs when reflecting them. Thus, in order to achieve a test verdict, it is necessary to obtain the taint information associated with "how the browser parsed the output". A candidate solution is to obtain taint information up to the browser parse tree. However, since we are in a black-box context and want to find XSS exploits for real-world browsers, we cannot propagate the taint as [Sekar 2009] did with his home-written browser. Thus we propose to use taint inference techniques to obtain this information.

We illustrate these choices in Figure 3.4, and develop this second part of our approach in Chapter 5.

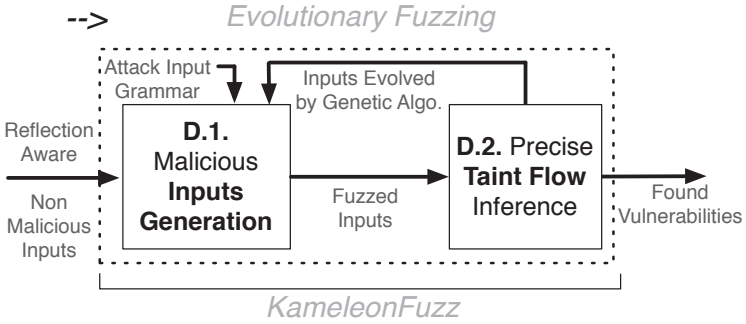


Figure 3.4: KameleonFuzz: Evolutionary XSS Fuzzing

Example We describe the execution of LigRE+KameleonFuzz on P0wnMe (page 18). We here focus on KameleonFuzz, once LigRE has inferred the control+taint flow models and generated input sequences.

Figure 3.2 contains a reflection for the value `2_e_g_a_s_sem` of the parameter *message*2 sent in the transition $7 \rightarrow 33$. An extract of the output O_{dst} is: `<input name="message2" value='2_e_g_a_s_sem'/>` where we highlight the reflection. Here, the reflection *context* is *inside* a tag attribute value. The context influences how an attacker generates fuzzed values. Listing 3.1 shows the server sanitizer for this reflection. It blocks simple attacks. Attackers search a fuzzed value s.t. if passed through the sanitizer, then its reflection is not syntactically confined in the context [Su & Wassermann 2006] i.e., it spans over different levels in the parse tree.

```
1 <?php function webapp_filter($str) {
2     if(eregi('"|`|>|<|;|/',$str)) {
3         $filtered_str = "XSS attempt!";
4     } else {
5         $filtered_str = str_replace(" ","",$str);
6     }
7     return $filtered_str;
8 } ?>
```

Listing 3.1: A vulnerable sanitizer in P0wnMe

Table 3.1 shows fuzzed values sent by w3af [Riancho 2011], a black-box open source scanner, when testing P0wnMe . W3af iterates over a list of fuzzed values. It does not learn from previous requests, nor considers the reflection context. As a result, all fuzzed values in Table 3.1 were affected by the filter described in Listing 3.1, and w3af considered this reflection not to be dangerous (false negative in vulnerability detection). Only the input value composed of characters having no special meaning in HyperText Markup Language (HTML) or JS (i.e., $\in [\![a-Z]\!]$) were not filtered. We illustrate in the following table the only reflection that w3af obtained.

Fuzzed Value (x_{src})	**Reflection**
SySlw	SySlw
uI<hf>hf"hf'hf(hf)uI	XSS attempt!
</A/style="xss:exp/**/ression(fake_alert('XSS'))">	
'';!--"<klqn>=&{()}	
<IFRAME SRC="javascript:fake _alert('klqn');"></IFRAME>	

Table 3.1: w3af fuzzed values (extract)

The chopping (step C of LigRE, illustrated in Figure 3.3) produces input sequences containing at least one reflection.

In step D.1, KameleonFuzz generates individuals, i.e., input sequences in which it fuzzes the reflected value by replacing the input parameter value by a word generated from the Attack Input Grammar (AIG). For each individual, the corresponding outputs are recorded and the taint is inferred between the fuzzed input value and the concrete output, but also between the tainted substrings of the concrete output and the nodes of the browser parse tree. This taint aware tree is an input for the test verdict (did this individual trigger an XSS?) and the fitness score (how close is this individual to triggering an XSS?). The best individuals are genetically recombined while still conforming to the AIG to create the next generation: e.g., the individuals 3 and 4 of generation 1 produce the individual 1 of generation 2. This process is iterated until a tester defined stopping condition is satisfied (e.g., one XSS is found). Table 3.2 illustrates this evolution.

Fuzzed Value (x_{src})	**Reflection**	XSS	Fit.	Gen.
`T9nj1'><script>alert (18138)</script>`	`XSS␣Attempt!`		3.1	1
`oH1eqL'␣onload=" document.body.inner HTML+='<div␣id=90480> </div>'"␣fakeattr='`	`XSS␣Attempt!`		3.2	1
`ZuIa2'␣onload =alert(94478)`	`ZuIa2'onload =alert(94478)`		13.3	1
`WUkp'\tLgpRa`	`WUkp'\tLgpRa`		9.1	1
`WUkp'\t␣onload=' alert(94478)`	`WUkp'\tonload ='alert(94478)`	✓	18.5	2

Table 3.2: KameleonFuzz fuzzed values (extract) of the reflection (t_{src} = 7→33)(*message*)→(t_{dst} = 7→33)

The sanitizer in Listing 3.1 removes the space ␣, but not \t,\r or \n. An extract of the output o_{dst} for the last individual is
`<input name="message2" value='WUkp'\t onload='alert(94478)'/>`

Using string edit distance and a threshold, the taint is inferred between the tainted substrings of o_{dst} and each node of the parse tree obtained from a browser. This produces a Taint-Aware Tree (TAT), as illustrated in Figure 3.5.

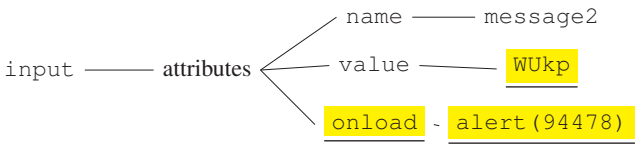


Figure 3.5: A Taint-Aware Tree (TAT) TT_{dst} (extract). The payload is a message box that displays 94478 (harmless).

Figure 3.6: One Taint-Aware Patterns (TAP), represented in a Linear Syntax (a tainted event handler attribute)

The TAT are filtered, using a set of Taint Aware Patterns (TAP) (Figure 3.6). Each TAP is characterized by a non-confinement of a tainted value. TAP are provided by the tester, who can use a very generic TAP, or, for example, use ones

which only detect XSS exploits triggering the JS interpreter. Since the TAP in Figure 3.6 matches the TAT in Figure 3.5, the syntactic confinement of the reflection of x_{src} is violated and the individual is a successful XSS exploit.

This example illustrates how evolutionary input generation can adapt to sanitizers.

CHAPTER 4

Web Application Model Inference for Black-Box XSS Detection

> We think too small, like the frog at the bottom of the well.
> He thinks the sky is only as big as the top of the well.
> If he surfaced, he would have an entirely different view.
>
> Mao Zedong

XSS involve both control and taint flows, as they rely on an input value being partly copied to a transition output. Thus, our approach for automatically detecting XSS in black-box consists of two steps: "crawling" infers the *control and taint flow* of the application, then "fuzzing" generates malicious inputs to exhibit vulnerabilities. In the current chapter, we focus on the first step, components A, B and C of Figure 4.1. This chapter addresses the problems of Section 2.5.1 (see page 33): *XSS.1 How to navigate in the Application? XSS.2.1 Where are the potential sinks? XSS.3.1 Where to fuzz inputs? XSS.3.3 How to prioritize inputs fuzzing?*

4.1 Our Approach

4.1.1 High Level Overview

We propose LigRE, a reverse-engineering tool which produces a model used to guide the fuzzing towards detecting XSS vulnerabilities. As illustrated in Figure 4.1, it first learns a control plus taint flow model, and then generates slices of this model to guide the fuzzing.

During step A of Figure 4.1, LigRE learns the *control flow* of the application, using a state aware crawler, to maximize coverage. During step B, LigRE annotates the inferred model with observable *taint flows* of input values into outputs to produce a *control plus taint flow model*. Annotations flow from a source t_{src} to a potential sink t_{dst}. We use an heuristic driven substring matching algorithm for its efficiency and as filters impact is generally low on reflections on non malicious input parameters values.

After step B, we prioritize the most promising annotations. For each of them, step C produces a slice of the model. Slices are chopped models. They permit to drive the application to the origin of t_{src}, for sending a malicious value x_{src}, and then to produce inputs guiding a fuzzer to navigate towards t_{dst}, for observing the

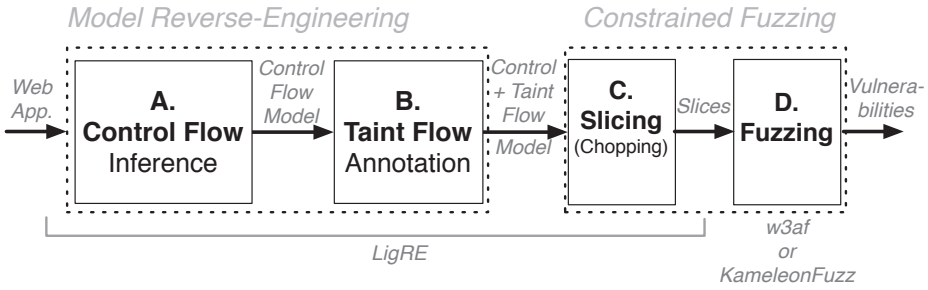


Figure 4.1: High Level View of our Approach

effects of x_{src}. The fuzzing, step D, consists in creating and prioritizing values x_{src} depending of their effect. The fuzzing step will be described in Chapter 5.

4.2 Control Flow Inference

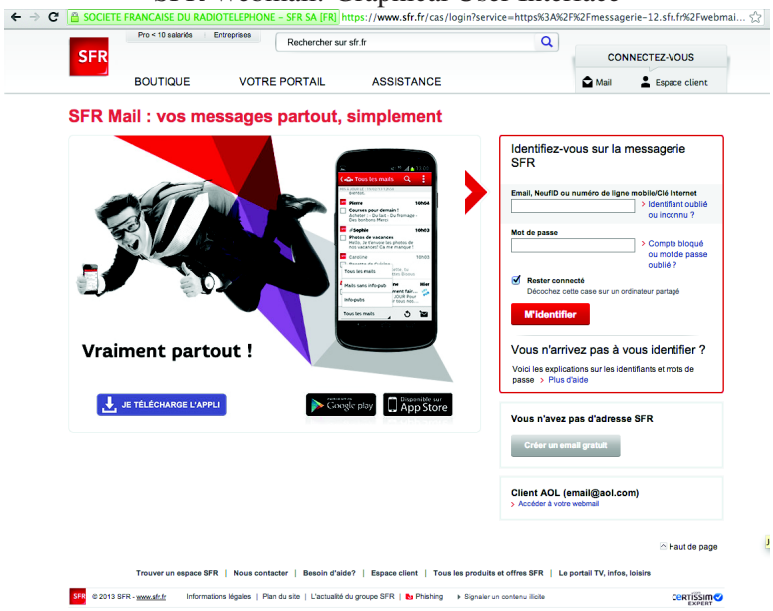
Step A in Figure 4.1 takes as input the description of a remote web application (e.g., interface, authentication credentials), interact with it, and outputs a *control flow model* (CFM). A CFM formalizes the observable behavior of a web application in a black-box test context. Nodes (states) represent webpages and transitions represent requests and associated responses. However, storing only the aforementioned information is not sufficient for constructing a precise model since the web application may have internal *macro-states* (e.g., new user created, logged-in, logged-out, etc.). We capture this notion of macro-states by adding colors to nodes, as in [Doupé *et al.* 2012].

The control flow inference step uses heuristics to identify which request changed the macro-state (Section 4.2.3), chooses the next request to be performed (Section 4.2.4), and assess the degree of certainty in the model (Section 4.2.5.1). The model is iteratively built.

Non-Deterministic Values (NDV) In the process of `abstraction`, some parameters are omitted: NDV a.k.a. nonces [Wikipedia 2006], i.e., output parameters whose values differ when sending twice a given input sequence (and resetting the system in between) and which may be used in subsequent inputs. Examples of NDV include: anti-CSRF tokens [OWASP 2013a], session_id stored in cookies [Barth 2011], view_states [Microsoft 2004]. In the presence of NDV, crawlers achieve a limited coverage. More important, since we are interested in building a Control Flow Model of the tested application, the presence of NDV may change the abstracted output, thus resulting in a state explosion, whereas the execution reached a previously encountered state.

We address this problem by requiring the human tester to identify NDV. In order to do so, the tester has to observe which parameter values in a page model

SFR Webmail: Graphical User Interface



	Execution №0	Execution №1

HTTP Reply (extract)	`<div class="box boxMire">`
	`<form action="/cas/login?domain=webmessagerie-pub&service=https%3A%2F%2Fmessagerie-12.sfr.fr%2Fwebmail%2Fj_spring_cas_security_check" name="loginForm" id="loginForm" method="post">`
	`<input type="hidden" name="lt" value="_cEA2E9A07-5C78-0994-1282-6A572358D104_k2D37F6BC-38DB-F171-2701-711C42E68280" />`
	`<input type="hidden" name="_eventId" value="submit" />`
	`<h2>Identifiez-vous sur la messagerie SFR</h2>`
		`<div class="box boxMire">`
		`<form action="/cas/login?domain=webmessagerie-pub&service=https%3A%2F%2Fmessagerie-13.sfr.fr%2Fwebmail%2Fj_spring_cas_security_check" name="loginForm" id="loginForm" method="post">`
		`<input type="hidden" name="lt" value="_cA3686786-48C9-185C-9262-C78FC3B9E59D_k62759BC4-C6D0-1319-FDC9-AEED15F8B01F" />`
		`<input type="hidden" name="_eventId" value="submit" />`
		`<h2>Identifiez-vous sur la messagerie SFR</h2>`

Page Model (extract)	page | /html/body/div/form | /cas/login | (domain,...,lt,...) | ('webmessagerie-pub',...,'**_cEA2E9A07-...**',...) | (GET,...,POST,...)	page | /html/body/div/form | /cas/login | (domain,...,lt,...) | ('webmessagerie-pub',...,'**_cA3686786-...**',...) | (GET,...,POST,...)

Figure 4.2: Visually Spotting a Nonce for the Parameter **lt** in SFR WebMail

are constantly changing when executing the exact same input sequence after an application reset.

For instance in Figure 4.2, when the tester submits twice the same input `GET /`, while resetting the application in between, the part of the page model corresponding to the parameter `lt` has a different value the second time, thus `lt` is a nonce. Thus, for each NDV name, the tester has to execute twice at least one given transition. [1]

For one given transition, detecting NDV may seem easy according to the previous example, assuming the knowledge of a complete control flow model. However, during the control flow inference, the complete model is not yet available and is being inferred. Determining both the control flow model and the NDV are two connected problems: not identifying NDV may lead to parts of the model being duplicated, and identifying NDV requires the ability to navigate to a transition which contains one. However, our inference algorithm is not able to solve them both simultaneously. Thus we require the tester to identify NDV.

Tester Provided Values The tester may also provide values for extending the page models and thus generating new spiderlinks (e.g., a special login value for a field named `uid` in a specific DOM path).

4.2.1 Overview

Algorithm 4.1 shows the inference of a control flow model from a web application. The control flow inference step infers partial control flow models (not necessarily completely specified for each input).

Until a tester defined multi-criterion stopping condition is met (e.g., number of requests, duration, number of different pages seen, number of macro-states, etc.), LigRE iterates the following process (line 8).

LigRE resets the web application to its initial state using a tester written script. The first spiderlink to be chosen is the start input (generally `GET /`). LigRE sends the concretization of the current spiderlink to the web application, and abstracts the corresponding application output (HTTP reply) to a page model.

LigRE then consults the history plus the current spiderlink and page to determine if the macro-state has changed since the last time the same spiderlink was sent (line 15). If this is the case, LigRE determines which spiderlink in the history changed the state (line 17), thanks to the score heuristic (see Section 4.2.3). The identifiers of each history entry are updated, and the colors of the macro-states are computed (line 21 consists in merging identical macro-states).

LigRE updates the history by storing the spiderlink and the page model. LigRE then updates the control flow model w.r.t. the new information in the history (line 25).

[1] In this example, the form submission method is POST, and the url to submit the form contains parameters which will be sent using the GET method.

A new spiderlink is chosen in the ones available in the current page model. If none is available, or if the current input sequence is longer than the maximal length allowed by the tester, LigRE resets the web application, and builds a new sequence following the aforementioned process by choosing the start spiderlink. The exploration is thus a Depth First Search (DFS), until a contradiction is detected, or a sequence of maximal length has been produced.

Similarly, when a contradiction is detected for a given macro-state, and the confidence of the color not already chosen for the macro-state is higher than the one in the control flow model, then we backtrack, undoing the latest macro-state change, and reset the application and start a new input sequence.

LigRE makes use of heuristics, because the problem of determining macro-state is addressed on the fly during the navigation problem.

Algorithm 4.1: Control Flow Inference

```
1  # IN: nonces, webapp, stopping_criterion
2  # OUT: cfm
3  history=[]
4  webapp.reset()
5  curr_sequence_length = 0
6  curr_identifier = 0
7  spiderlink = config.start_spiderlink
8  while(not stopping_criterion):
9      if(curr_sequence_length>MAX_SEQUENCE_LENGTH):
10         webapp.reset()
11         curr_sequence_length = 0
12         spiderlink = config.start_spiderlink
13     output = webapp.send(spiderlink.concretize(nonces))
14     page = output.abstract(nonces)
15     if(cfm.macro_state_changed(spiderlink,page,history)):
16         curr_identifier += 1
17         k = cfm.index_changed_macro_state(spiderlink,page,history)
18         for i in range(k,len(history)):
19             history[i].identifier = curr_identifier
20         page.identifier = curr_identifier
21         cfm.compute_colors(cfm.identifiers,cfm.pages)
22     else:
23         page.identifier = curr_identifier
24     cfm.history.append({spiderlink,page})
25     cfm.update_hist(history)
26     spiderlink = page.pick_spiderlink()
27     curr_sequence_length += 1
28 return cfm
```

4.2.2 Control Flow Notions

The history of inputs and outputs serves to build a *navigation tree*, which is used to build a *CFM*. Both are colored. Their coloration evolves to characterize the *macro-states*.

4.2.2.1 Macro-State

Macro-state is an important notion for understanding the control flow of a web application. It designates "anything that influences the executed code at server side" [Doupé *et al.* 2012]. Both nodes and macro-states represent the current execution context of the web application. They differ in their granularity. A node is characterized by a page (i.e., the last output). Whereas a macro-state is a set of nodes, i.e., at a higher level of abstraction, and is characterized by a common behavior of these nodes. Definition 10 formalizes this notion.

Definition 10 Macro-State

Let $W = (N, n_0, T, I, P, \Pi)$ be a Web Application (Definition 4).

A macro-state is a set of nodes which is coherent w.r.t. its successor nodes.

Let $C \subset \mathbb{N}$ be a set of colors. Let $col : N \to C$ be a coloring function which associates a color to each node. We say that col is a valid macro-state coloring iff, for any $n_a, n_b \in N$, $col(n_a) \neq col(n_b)$ whenever any of the following conditions hold:

- $\exists i \in I, \exists (n_c, n_d) \in N^2$ s.t. $(n_a, i, n_c) \in T \wedge (n_b, i, n_d) \in T \wedge \Pi(n_c) \neq \Pi(n_d)$
- $\{\Pi(u) | \exists i \in I, (n_a, i, u) \in T\} \cap \{\Pi(u) | \exists i \in I, (n_b, i, u) \in T\} = \varnothing$

If col is such a coloring, for each $c \in C$, the set of nodes $N_c = \{n \in N, col(n) = c\}$ is a macro-state.

4.2.2.2 Control Flow Model (CFM) (`model`)

A CFM is a Web Application with colors (i.e., macro-states). It is defined in Definition 11. The nodes and continuous arrows of Figure 3.2 are an example of CFM. The inferred CFM are not necessarily completely specified for each pair of node and input.

Definition 11 Control Flow Model (CFM)

A CFM is a 8-uple $M = (N, n_0, T, I, P, \Pi, C, col)$ where $W = (N, n_0, T, I, P, \Pi)$ is a Web Application (Definition 4) and $col : N \to C \subset \mathbb{N}$ is a coloring of W s.t. the macro-states partition N:

- for each color $c \in C$, let N_c be the set of nodes in N having this color. Either N_c is empty, or N_c is a macro-state (Definition 10)

4.2.2.3 Navigation Tree (`history`)

The Navigation Tree (`history` in Algorithm 4.1) is a set of traces. It is a prefix tree which contains the sequences of abstract inputs (spiderlinks) and outputs (page models). This navigation tree is an auxiliary structure for building the CFM.

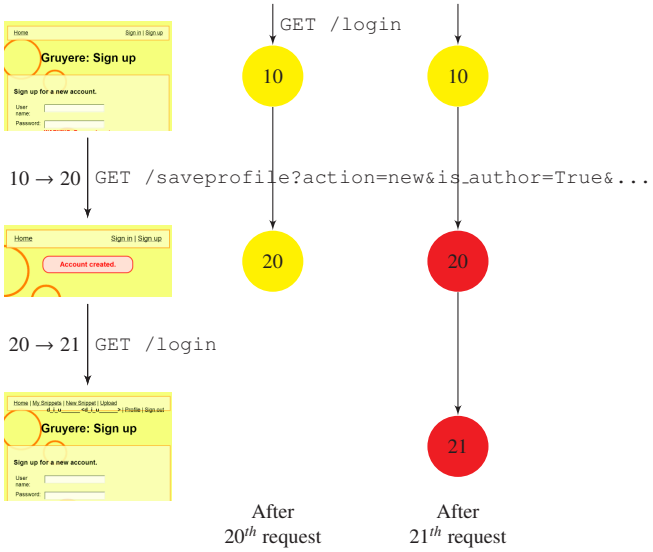


Figure 4.3: Evolution of the Navigation Tree when the Macro-State Changes

4.2.3 Macro-State Change Detection

As the stopping criterion will halt exploration before the Web Application is fully explored, our inferred model is not completely specified. This holds for the step A (control flow inference), but also for the step B (taint flow inference).

We build the CFM iteratively in step A, using a DFS exploration, so we need to characterize the current macro-state after each request. In order to do it, we need to address four sub-problems: Did the macro-state change? Which request changed the macro-state? What is the current macro-state? Which link to pick next?

In order to alleviate the computational complexity, we address these sub-problems using heuristics, inspired from [Doupé *et al.* 2012]. We added parameters in Table 4.1 and Table 4.2, and adjusted their weights using results from experimentation by observing which combinations increased the efficiency of the control flow inference. **??** details the dimensions and the rationale behind each dimension.

4.2.3.1 Example

Figure 4.3 shows the evolution of an extract of the navigation tree (`history`) when a macro-state change occurs. In this example, the spiderlink `GET /login` permits detecting the macro-state change, because the page model obtained in → 10 is different from the one obtained in 20 → 21, and the same spiderlink `GET /login` was executed. `index_changed_macro_state` selects 10 → 20 as the cause of the state-change, because it has the highest `score` (see Table 4.1) value among [→ 10, 10 → 20, 20 → 21].

+ or - weight	id	dimension name
++	1	number of input parameters
+	2	distance between page models ($p_{prev} \underrightarrow{i} p_i$)
+	3	HTTP method
−	4	number of times performed (total)
−	5	number of times it changed the state
−	6	number of requests between i and i_{detect}
−−	7	number of potential contradictions (approx.)

Table 4.1: Dimensions of the `score(spiderlink i)` heuristic

4.2.3.2 Did the macro-state change? (`macro_state_changed`)

If a spiderlink i is sent twice to the application during the requests *prev* and *detect*, and the obtained page models are different (i.e., $(o_{prev} = i_{prev}$`.concretize().send()).abstract()` $\neq (o_{detect} = i_{detect}$`.concretize().send()).abstract()`), then the macro-state changed. This is the case for the spiderlinks i_{prev}=`GET /login`($\rightarrow$ 10) and i_{detect}=`GET /login` (20 $\rightarrow$ 21) in the navigation tree extract shown in Figure 4.3.

4.2.3.3 Which request changed the macro-state? (`index_changed_macro_state`, `score` heuristic)

If a macro-state change is detected between i_{prev} and i_{detect}, then the question "which request in the history between those changed the macro-state?" arises. To answer it, the heuristic function `score` represents the likelihood of a request having changed the macro-state. For a spiderlink $i \in [i_{prev}, \ldots, i_{detect}]$ the higher the value of `score(i)`, the more likely i changed the macro-state. The dimensions of `score` are listed in Table 4.1. If there is a +, resp. −, in front of the dimension, then `score` is increasing, resp. decreasing, w.r.t. this dimension. `score` is used in `index_changed_macro_state` in Algorithm 4.1.

The following dimensions compose the heuristic of Table 4.1:

- **1. Number of input parameters**: the more inputs parameters for a given spiderlink, the more likely it will change the macro-state (e.g., when creating a new user in an application).

- **2. Distance between page models**: the more distinct are the page models of i and i_{prev}, the more likely they correspond to nodes in different macro-states. Due to its effectiveness, we use the PQ-gram distance for measuring the similarity between page models [Augsten *et al.* 2005].

- **3. HTTP method** : a POST method is more likely to change the macro-state than a GET method, thus it will have a higher score.

- **4. Number of times performed (total)**: number of times that this spiderlink has already been sent. Our navigation strategy prefers request that permit determining a macro-state change, but not request that actually do change the macro-state. Thus a frequently sent spiderlink is not very likely of having changed the state.

- **5. Number of times this spiderlink changed the state** : since we are using heuristic functions, we want to be error tolerant, that is if once we determined that a given spiderlink changed the macro-state, we want to decrease the likelihood of repeating this error.

- **6. Number of requests between i and i_{detec}**: since our navigation strategy prefers requests that help determining a macro-state change, it is likely that we detected a macro-state change early, that is that i_{detect} is probably close of the spiderlink that changed the macro-state.

- **7. Number of potential contradictions (approximative)**: if we hypothesize that i changed the macro-state, how many potential contradictions would we have? It is likely that choosing a wrong spiderlink as the reason for a macro-state change would increase the number of contradictions.

The final score is a weighted linear sum of each dimension.

4.2.3.4 What is the macro-state of the current node? (`compute_colors`)

The current node is the result of the submission of the spider-links since the last reset. In order to know if the current node is one previously encountered, it is necessary to merge macro-states.

For this purpose, an *identifier* is associated to each node. If the macro-state changes, then the current identifier is updated to a one different of the preceding node, it is unchanged otherwise (see Algorithm 4.1). [Doupé *et al.* 2012] reduced this macro-state collapsing problem to the coloring of an undirected graph of identifiers [Doupé *et al.* 2012]. If there is an edge between two identifiers (e.g., A and B), then they will have different colors, otherwise they will be merged (e.g., B and D identifiers are merged in the same color $B + D$ in Figure 4.4).

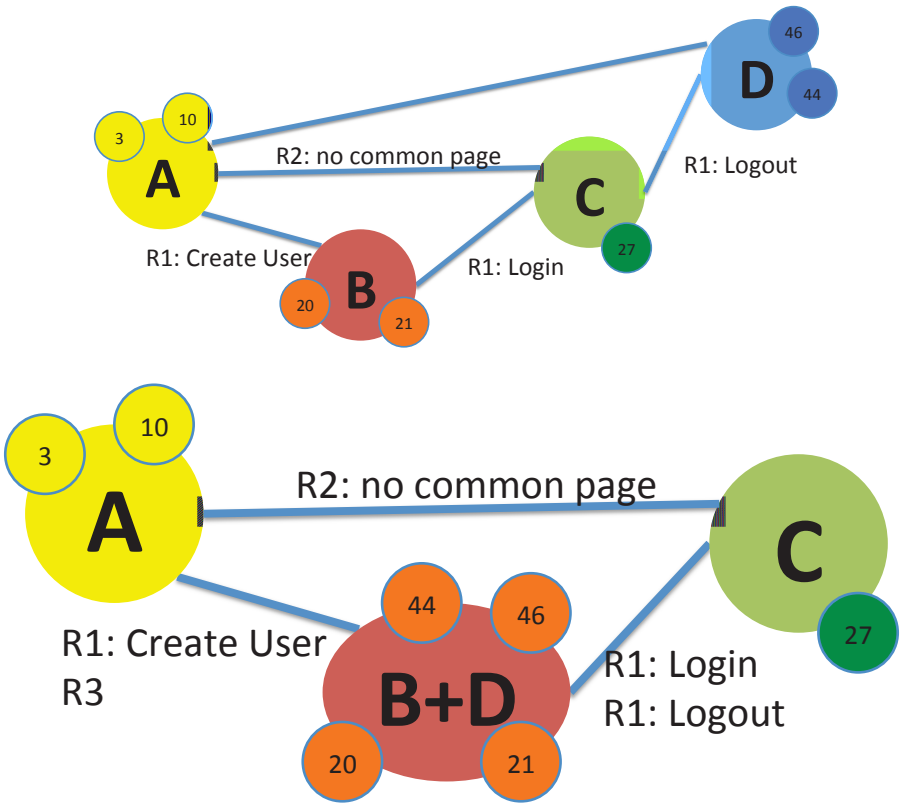


Figure 4.4: Identifiers Merge: B and D denote the same Macro-State (extract of the Google Gruyere Macro-State Coloring Process)

There are four rules to add an edge between two identifiers / macro-states α and β:

- Rule1: there is a macro-state change from $\alpha \rightarrow \beta$ or $\beta \rightarrow \alpha$ (e.g., $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$ etc.)
- Rule2: they have no common pages ($pages(\alpha) \cap pages(\beta) = \varnothing$) (e.g., A and C)
- Rule3: if $\exists$ a spiderlink i, and identifiers γ, δ, s.t.
  - by executing i when the application is in the macro-state $color(\alpha)$ drives it to the macro-state $color(\gamma)$: $\alpha(i) \rightarrow \gamma$
  - and by executing i when in the macro-state $color(\beta)$ it leads to a macro-state $color(\delta)$: $\beta(i) \rightarrow \delta$
  - and the reached macro-states are different: $color(\gamma) \neq color(\delta)$
  - ... then it means that α and β should not be mapped to the same macro-state: $color(\alpha) \neq color(\beta)$

- Rule4: adding an edge reduces the number of potential contradictions

Backtracking may occur during coloring (see Section 4.2.5.3).

Algorithm 4.2: Compute Colors

```
# IN: history, cfm, identifiers
# OUT: cfm, identifiers

def compute_colors():
    cfm.compute_rules_1_and_2()          # R1: same input, different page model
                                         # R2: no common page model
    cfm.compute_rule_4()                 # R4: contradictions
    identifiers.greedy_coloring()
    while(cfm.compute_rule_3() > 0 ):    # R3: same input leads to different macro-states
        identifiers.greedy_coloring()

class CFM:
  def compute_rules_1_and_2():
    for a in identifiers:
        for b in identifiers:
            next_node_b = false
            if(a != b):
                at_least_one_common_page_model=false
                for t_a in a.transitions:
                    for t_b in b.transitions:
                        if(t_a['input']==t_b['input']):
                            # rule 1: same input, different page model
                            if(t_a['node']['page_model']!=t_b['node']['page_model']):
                                identifiers.edges.append([a,b])
                                next_node_b = true
                                break
                        if(t_a['node']['page_model']==t_b['node']['page_model']):
                            at_least_one_common_page_model=true
                    if(next_node_b):
                        break
                # rule 2: no common page model
                if(not at_least_one_common_page_model):
                    identifiers.edges.append([a,b])
```

Algorithm 4.3: Compute Colors (cont.)

```
34    # R4: contradictions
35    def compute_rules_4():
36      for a in identifiers:
37          for b in a.contradiction_observed:
38              identifiers.edges.append([a,b])
39    # R3: same input leads to different macro-states
40    def compute_rule_3():
41      num_of_added_edges=0
42      for a in identifiers:
43          for b in identifiers:
44              if((a != b) and ([a,b] not in identifiers.edges) and ([b,a
                    ] not in identifiers.edges)):
45                  for t_a in a.transitions:
46                      next_b = false
47                      for t_b in b.transitions:
48                          if(t_a['input']==t_b['input']):
49                              if(t_a['node'].identifier.color != t_b['
                                    node'].identifier.color):
50                                  identifiers.edges.append([a,b])
51                                  num_of_added_edges +=1
52                                  next_b = true
53                                  break
54                      if(next_b):
55                          break
56      return num_of_added_edges
57
58  class Identifier:
59    def greedy_coloring():
60      curr_color = -1
61      for a in identifiers:
62          neighbors_colors=[]
63          for b in a.get_identifiers_edges():
64              if(b.color != -1):
65                  neighbors_colors.append(b.color)
66          neighbors_colors.sort()
67          # use the most recent available color
68          for j in range(curr_color,-1,-1):
69              if((found_color==-1) and (j not in neighbors_colors)):
70                  found_color = j
71                  break
72          if(found_color==-1):
73              current_color +=1
74              found_color = current_color
75          a.color = found_color
```

4.2.4 Navigation Strategy

Each time LigRE receives an output of the web application, it abstracts this concrete output to a page model and update the macro-states colors. Since we perform a DFS exploration, LigRE chooses the next spiderlink to explore. Depending on the tester parameters, it may generate additional spiderlinks – than the ones present in the page model – for facilitating future taint flow inference.

4.2.4.1 Choosing the Next Spiderlink to Explore (`pick_spiderlink`)

After obtaining a page model p, LigRE must decide what is the next spiderlink in $spiderlinks(p)$ to explore. The heuristic function `navigating` represents the likelihood of a spiderlink to be chosen as the next one to be executed on the application. For a given spiderlink i, the higher the value of `navigating(i)`, the more likely i will be picked. Table 4.2 lists its dimensions. `pick_spiderlink` in Algorithm 4.1 uses it.

+ or - weight	**id**	**dimension name**
+ + +	1	request never executed
++	2	(1+consecutive_contradictions)*num_state_change
++	3	num recently sent
++	4	number of artificially generated parameter values
+	5	number of times sent
−	6	spiderlink_method_weight
−	7	number of times it changed the macro-state

Table 4.2: Dimensions of the `navigating(spiderlink i)` heuristic

The dimensions of Table 4.2 model the several intuitions:

- **1. Request Never Executed**: we want to increase the knowledge of the application.
- Some spiderlinks may be more likely to be picked up, as they permit detecting a state change (**2. (1+consecutive_contradictions)*num_detect_state_change**). However, we do not want that only those are choosen, thus we temporarily introduce a penalty if they have been recently picked (**3. Number of Times Recently Sent**).
- **5. Number of Times Sent**: we want to favor less explored spiderlinks.
- **4. number of artificially generated parameter values**, **6. spiderlink_method_weight** and **7. number of times it changed the macro-state**: we want to explore as much as possible of the current macro-state before exploring the next one. Thus, since POST requests are statistically more likely

to change the macro-state than GET requests (e.g., user creation, user login), navigating is a decreasing metric w.r.t. this dimension.

As illustrated in Algorithm 4.4, either the current node contains unexplored spiderlinks and one of them is chosen according to their `navigating` score, or the shortest path in the model to nodes containing non explored spiderlinks is computed using [Dijkstra 1959]'s algorithm.

Algorithm 4.4: Pick Spiderlink

```
1
2  # IN: cfm, page, history
3  # OUT: spiderlink
4
5  def pick_spiderlink(cfm, page, history):
6      chosen=None
7      never_explored = []
8      for sp in page.spiderlinks:
9          if(sp not in history):
10             never_explored.append(sp)
11     if(len(never_explored)>0):
12         never_explored.sort(key=lambda i: navigating(i),reversed=true)
13         sp_set = never_explored
14     else:
15         explored_n_times = []
16         n = 1
17         while((len(explored_n_times)==0) and (n<config.stop.
               N_MIN_LINKS)):
18             explored_n_times = cfm.
                   get_all_transitions_explored_n_times(n)
19         if(len(explored_n_times)==0):
20             raise Exception('cfm built')
21         explored_n_times.sort(key=lambda i: dijkstra(page.
               current_spiderlink,i))
22         sp_set = explored_n_times
23
24     chosen = sp_set[0]
25     return chosen
```

The stopping criterion evaluates to true when for each node of the CFM, the outgoing transitions have been explored a tester defined number of times. The tester can limit the number of requests and the execution time. In our experiments, we limit the number of sent requests. We adjust this metric by iteratively, browsing manually the application, setting a limit, inferring a control flow model, observing

the obtained CFM, and eventually adjusting the number of requests. We adjusted this metric depending on the tested web application (e.g., for P0wnMe , we limit to 60 requests, whereas for Gruyere, we limit to 200). Adjusting this metric is a trial and error process, which converged around 25 iterations for the first tested applications to around 8 for the most recently tested ones.

4.2.4.2 Pruning

Testers may want to prune the model for readability, speed, or desire to concentrate the testing effort in one part of the application. This process is known to reverse engineers of binary executables [Guilfanov 2008]. LigRE permits to specify pruning patterns in order not to explore matching spiderlinks.

In Figure 4.5, we illustrate one pruning pattern. A pruning pattern matches DOM tree nodes, in order to prevent LigRE to build spiderlinks for the <A> or <FORM> DOM nodes containing in the subtrees matched by this pattern.

In Figure 4.6 we illustrate the impact that pruning has on the obtained CFM, specifically for WebGoat [OWASP], a deliberately vulnerable JSP web application (see Table 6.1 in page 91). We want to focus the testing on one particular WebGoat "lesson": "Stored XSS".

We define pruning patterns manually specifically for each web application we want to test.

```
value="A_TEXTNODE Stage 1: Bypass Business Layer Access Control" />
value="A_TEXTNODE Stage 2: Add Business Layer Access Control" />
value="A_TEXTNODE Stage 3: Bypass Data Layer Access Control" />
value="A_TEXTNODE Stage 4: Add Data Layer Access Control" />


value="FORM_TEXTCONTENT_REGULAR_EXPRESSION (.*)Welcome to WebGoat !!(.*)" />


value="A_TEXTCONTENT_REGULAR_EXPRESSION &lt;img src=&quot;images/buttons/params.jpg&quot;(.*?)&gt;"/>
value="A_TEXTCONTENT_REGULAR_EXPRESSION &lt;img src=&quot;images/buttons/cookies.jpg&quot;(.*?)&gt;"/>
value="A_TEXTCONTENT_REGULAR_EXPRESSION &lt;img src=&quot;images/buttons/plans.jpg&quot;(.*?)&gt;"/>
value="A_TEXTCONTENT_REGULAR_EXPRESSION &lt;img src=&quot;images/buttons/hint(.*).jpg&quot;(.*?)&gt;"/>
```

Figure 4.5: Nine Examples of Pruning Patterns for Spiderlinks

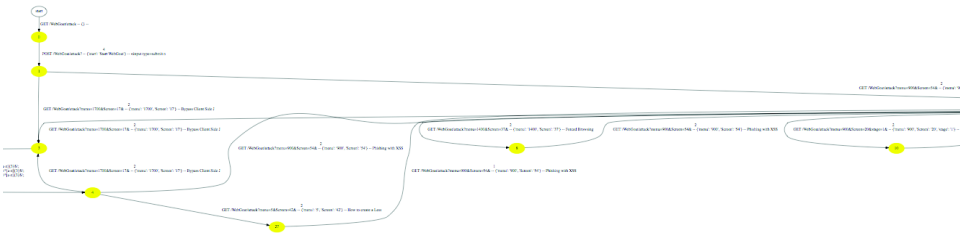
4.2.4.3 Artificial Spiderlink Creation

Depending on tester defined configuration, LigRE may create spiderlinks that contain artificially generated values (i.e., which are not present in the page models). This value creation aims at limiting the risks of collisions and false positives during the later taint flow inference.

We want a function *artif* that receives an input parameter *name* (e.g., *msg* in Figure 3.2) and produces a value s.t. the following properties hold for "most" input parameter names:

- it is easy to compute *artif(name)*

First LigRE Run: Initially Produced CFM after 60 requests: only one macro-state is detected, because only the very first transitions from the root node are explored.



Second LigRE Run: Produced CFM with pruning after 60 requests. As LigRE explore deeper parts of the application, it discovers a new (red) macro-state.

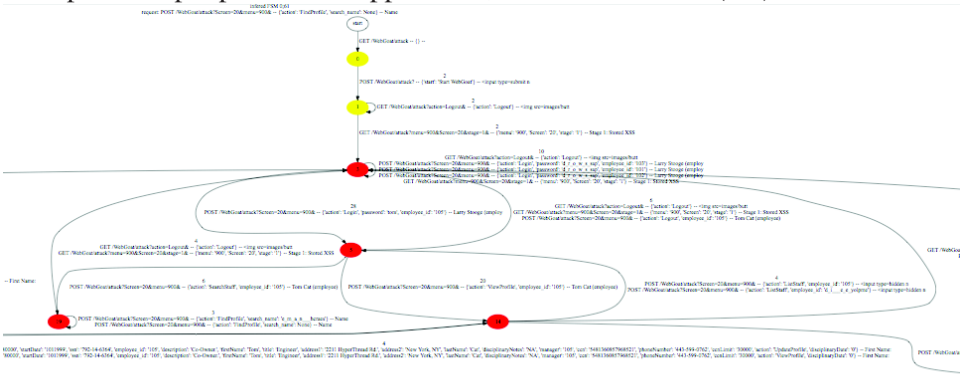


Figure 4.6: Result of Application of Spiderlink Pruning

- modifying an input parameter *name* significantly changes *artif(name)*

Those are two of the four properties of ideal cryptographic hash functions. Some web fuzzers use hash functions [epsylon 2012].

In our implementation, we use our own function which creates parameter values by reversing the parameter name and by alternating characters from the reversed input parameter name and an extension string (may be tester provided, we hardcoded default ones ; should not contain "special" HTML characters such as {', ", >, /}), as illustrated in Table 4.3. This has the advantage of permitting a human tester to visually identify the tainted parameter source. For two given input parameters having the same name, our technique voluntarily generates the same value, even if they originate from different transitions in the CFM.

Param. Name	Extension String	Generated Param. Value
pw	12_45_78_0	w1p2w_p4w5p_

Table 4.3: Automatic Value Generation for Helping Taint Flow Inference

4.2.5 Backtracking

Backtracking consists of undoing parts of the model and recomputing them with an additional constraint. It occurs when either a *potential contradiction* (Definition 12) is observed on part of the model with a low *confidence*, or when executing a spiderlink on the CFM leads to a different page model than the one observed in the Web Application. Backtracking is a part of the `model.update()` process (Algorithm 4.1).

4.2.5.1 Confidence

The *Confidence* expresses the level of trust in a part of the model. This metric is applicable to a node or a transition. The higher its value, the more confident we are in the coloring of the element. Table 4.4 contains the dimensions used in this function.

weight	dimension name
–	number of nodes in the shortest path from root
–	number of unexplored spiderlinks in the page model of n
–	... that have same hash as one which permit determining a macro-state change

Table 4.4: Dimensions of the `confidence(node n)` heuristic

4.2.5.2 Potential Contradiction

A potential contradiction indicate that we may have assigned a page model to the wrong macro-state coloring. It is defined in Definition 12. Let us assume that the

Definition 12 Potential Contradiction

Let n_a and n_b be two nodes $\in N$. A potential contradiction between n_a and n_b is defined as follows:

$$contradiction(n_a, n_b) = \begin{cases} True & \text{if } ((confidence(n_a) \neq confidence(n_b)) \\ & \wedge(page\_model(n_a) == page\_model(n_b)) \\ & \wedge(color(n_a) \neq color(n_b))) \\ False & \text{otherwise} \end{cases}$$

node b is the current state. If there exists a node a, s.t. $contradiction(a, b)$ is *True*, then we *may* have missed detecting a state change. Thus contradictions are inputs for `navigating` (see Table 4.2) and `score` (see Table 4.1).

4.2.5.3 Backtracking

We hypothesize that the web application is deterministic at the abstract level: if an input sequence of spiderlinks from the start node is executed several times, the sequences of obtained page models are the same.

Each sent spiderlink is executed on the application, and on the currently inferred CFM. It may happen that the CFM execution leads to a different page model than the application one. This is a non-determinism: either the application is not deterministic (and the tester missed a nonce in the abstraction process), or the current CFM is not correct. We assume it is the second case.

In such a situation, our heuristic assumes that the ultimate macro-state change was not correct: we considered the identifiers α and β to map to the same macro-state, but this turned out to be wrong. Thus, we add an edge between α and β (such edges are used in the `compute_rule_4()` of Algorithm 4.2), redo the coloring and update of the model, reset the application, and start a new sequence from the initial node.

4.3 Taint Flow Model Annotations

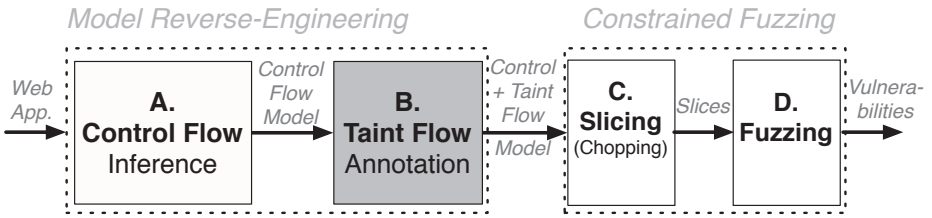


Figure 4.7: Step B: Taint Flow Inference

The taint flow model annotation corresponds to step B in Figure 4.1. It consumes a Control Flow Model (CFM), to which it adds inferred taint flows, thus producing a hybrid Control plus Taint Flow Model (CTFM), such as the one represented in Figure 3.2. In such a model, the bold text represents the source of a reflection t_{src}, and the blue/dotted arrow edges designate the reflection destination t_{dst}.

This step consists of first generating walks in the CFM, and then actively submitting those walks to the web application while inferring observable taint flows.

During this step, only taint flows are added to the CFM, no transitions are added or removed, even though a precise analysis may discover new macro-states. This is a choice of our implementation.

We compute the control flow inference (step A) and the taint flow inference (step B) separately. The reason is that until step A (control flow inference) is finished, we are unsure whether the execution context of the application leads to a node for which we already inferred the taint. Thus performing control and taint flow inference separately permits to only compute the taint when necessary. This

is important, as on the tested applications, the taint flow is inferred significantly more often than the control flow inference.

4.3.1 Definitions

Reflection Context and CTFM are defined respectively in Definition 13 and Definition 14. The taint-flow computation is explained in Section 4.3.3.

A reflection context is the output structure in which a reflected input value is supposed to be confined (Definition 6) during the processing of a non-malicious input. We list several reflection contexts in Table 4.5, and formally define this notion in Definition 13.

outside an HTML tag	`<h1>`
inside an HTML src/href attribute value	`<a href="  "/>`
inside a non src/href HTML attribute value	`<input value="  "/>`
inside an HTML textarea	`<textarea>  </textarea>`
inside a CSS value	`body { background-image: url(images/  .png); }`
inside a JS value	`var zipcode = "  ";`

Table 4.5: List of Considered Reflection Contexts (CTX) for G_O =HTML

Definition 13 Reflection Context

Let G be a grammar (e.g., HTML), CTX be a set of reflection contexts (Table 4.5), $o_{dst} \in \Sigma^*$ be a concrete output and a word of G, and $v \in \Sigma^*$ be a non-empty substring of o_{dst}.

The reflection context $ctx(v) \in CTX$ is the narrowest structure in G (non-terminal production rule) in which the reflected value is confined when sending non fuzzed values. If G is context-free, then $ctx(v)$ is the smallest word in G containing v and derived from one unique terminal.

4.3.2 Generating Walks

Random walk and Breadth First Search (BFS) are the implemented strategies for generating inputs from a CFM. The submission of those inputs is performed in a DFS manner. Since we generate identical values if the same parameter is present in several transitions, we want to reset the application frequently enough to avoid over-tainting. Our input sequence creation strategies limit the length of the input sequences, and the number of times the sequences traverse each node. If a sequence is a prefix of another one, then we only keep the latter. We analyzed XSS on

Definition 14 Control and Taint Flow Model (CTFM)

Let Σ be an alphabet, and G be a grammar. A Control and Taint Flow Model (CTFM) is composed of:

- a CFM (Definition 11)
- a taint-flow function df: $(\Sigma^+ \times T \times T) \rightarrow CTX^+$, s.t. for a reflection $refl = (x_{src}, t_{src}, t_{dst}, o_{dst})$ of the value x_{src} of a parameter $name \in \Sigma^+$, $df(name, t_{src}, t_{dst})$ produces the list of Reflection Contexts (Definition 13) of x_{src} into o_{dst} w.r.t. G

fifteen applications of various complexity, and observed that the longest shortest path between t_{src} and t_{dst}, both included, is 4 transitions, and the shortest path to reach the deepest t_{dst} was 8, thus we arbitrarily limit the length of the generated sequences to 8 (prefix+suffix).

4.3.3 Computing Taint Flows

For each sequence $I = (t_1, ..., t_k)$, for each concrete output o_j, $j \in [1..k]$, for each previously sent input parameter value x_m, $m \in [1..j]$, a distance between x_m and o_j is computed.

Specifically, the taint flow inference consists in first searching in the output o_j for exact substrings of x_m of a minimal length, marking those found substrings, clustering them, and then computing the edit distance [Levenshtein 1966] from x_m to the clusters. If this distance is lower than an empirically determined threshold, then a taint flow is annotated on the CTFM.

Algorithm 4.5: Compute Taint

```
1  # IN: cfm, webapp
2  # OUT: ctfm
3
4  def from_cfm_to_ctfm(cfm,webapp,config):
5      # generate input sequences
6
7      reg_exp=[]
8      # submit each input sequence
9      for inp_seq in sequences:
10         for k in range(0,len(inp_seq)):
11             try:
12                 reg_exp[k]
13             catch IndexException:
14                 reg_exp[k] = []
```

Algorithm 4.6: Compute Taint (cont.)

```
15              i_k = inp_seq[k]
16              o_k = webapp.submit(i_k.concretize(config.nonces))
17              taint_flow = []
18              for j in range(k,-1,-1):
19                  for l in range(0,len(inp_seq[j].params)):
20                      if(k not in taint_flow[j][l]):
21                          # search for exact substrings
22                          inp_param = inp_seq[j].params[l]
23                          try:
24                              reg_exp[j][l]
25                          catch IndexException:
26                              reg_exp[j][l] = [
                                    build_reg_exp_of_min_length(inp_param[
                                    'value'], config.min_taint_length),
                                    build_reg_exp_of_min_length(inp_param[
                                    'value'], config.min_taint_cluster)]
27                          if(reg_exp[j][l][0].search(o_k)):
28                              taint_flow[j][l] += [k]
29                              continue
30                          # filters may be in place
31                          if(matches = reg_exp[j][l][1].search(o_k)):
32                              # see next page
33  def from_cfm_to_ctfm(cfm,webapp,config):
34      # ...
35      # indexed here for readability
36      if(matches):
37          clusters = []
38          # cluster them
39          curr_clust = 0
40          for c in range(0,len(o_k)):
41              if(matches.char_c_is_tainted()):
42                  if(num_of_misses > config.max_chars_inside_clus):
43                      curr_clust += 1
44                  clusters[curr_clust] += o_k[c]
45                  num_of_misses = 0
46              else:
47                  num_of_misses += 1
48          # is any cluster sufficiently close? (server-size filter)
49          for clust in clusters:
50              if(edit_distance(in=inp_param['value'],out=clust) <=
                    config.max_edit_distance):
51                  taint_flow[j][l] += [k]
52                  break
53          ctfm = {'cfm':cfm,'taint_flows':taint_flow}
54      return ctfm
```

Algorithm 4.7: Compute Taint (cont.)

```
55      def build_reg_exp_of_min_length(my_str,my_len):
56          to_compile=val.substr(0,my_len)
57          for k in range(1,len(val)):
58              to_compile = "|"+val.substr(k,my_len)
59          return re.compile(to_compile)
```

4.4 Flow-aware Non Malicious Input Generation

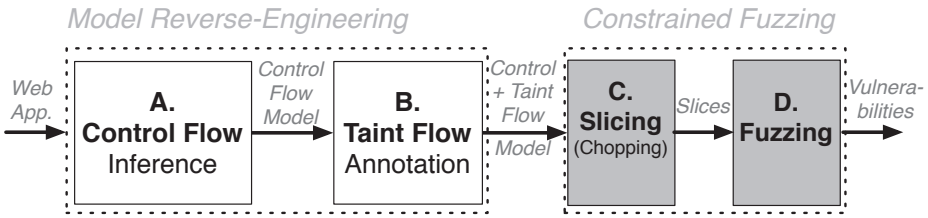


Figure 4.8: Step C and D: Chopping and Flow aware Fuzzing

4.4.1 Overview

Control+Taint Flow Aware Fuzzing encompasses steps C and D in Figure 4.1: first prioritizing the considered taint flows (Section 4.4.2), producing slices (Section 4.4.3), and then using those slices to guide a fuzzer. Its pseudo-code is in Algorithm 4.8. `get_reflections` returns the observed reflections by decreasing priority. It uses the `prioritization(reflection)` heuristic function whose dimensions are described in Table 4.6. The higher the value of `sum`, the more likely this reflection will be tested first. For each reflection, LigRE positions the application in the node from which t_{src} originates by sending a `prefix` sequence. Then LigRE feeds the fuzzer an authentication context (e.g., cookie) and a suffix obtained from the chopped model ($CH(t_{src}, t_{dst})$, see Section 4.4.3) for the fuzzer to navigate from t_{src} to t_{dst}.

4.4.2 Reflection Prioritization

Table 4.7 is an extract of the prioritization table in its initial state. dim_k corresponds to the dimension k of Table 4.6. Each tuple of cells $(t_{src}, x_{src}, t_{dst})$ line of Table 4.7 designates a reflection. Thus we have reflections a, b, c, d. Initially, `chosen_reflections`, the list of already chosen reflections, is empty. Since a has the highest prioritization value (see column *sum* in Table 4.7 and line 20 of Algorithm 4.8), a is the first reflection chosen. a is added to `chosen_reflections`. The dimensions of a are updated: $dim_{4,5}(a)$+=1. Then

Algorithm 4.8: Control+Taint Flow-aware Fuzzing

```
#IN: webapp, ctfm, fuzzer
#OUT: vulns

def control_data_aware_fuzzing(webapp, ctfm, fuzzer):
    vulns = []
    for refl in ctfm.get_reflections():
        webapp.reset()
        prefix=shortest_path(from=root,to=refl.src)
        webapp.execute(prefix)
        fuzzer.config.auth = webapp.context
        suffix = shortest_path(refl.src,refl.dst)
        fuzzer.config.urls = suffix
        vulns += fuzzer.do()
  return vulns

class CTFM(Object):
    reflections=[]
    def get_reflections(self):
        chosen_reflections=[]
        prioritization_table.init()
        while(len(chosen_reflections) < max_input_to_fuzz):
            chosen_index=-1
            prioritization_table.sort(key=lambda refl:( -refl.sum,
                refl.times_chosen))
            index_having_same_sum=0
            for i in range(1,len(prioritisation_table)):
                if(prioritization_table[0].times_chosen <
                    prioritization_table[i].times_chosen):
                    break
            index_having_same_sum=i-1
            chosen_index = random.randint(0,index_having_same_sum)
            chosen_reflections.append(reflections[chosen_index])
            prioritization_table.update_dimensions()
        return chosen_reflections
```

b is chosen, similarly: $dim_{4,5}(b)$+=1. Later on, either c or d will be chosen. Let us assume d is chosen first. Then $dim_{4,5}(d)+ = 1$ and $dim_4(c)+ = 1$ are updated, since c and d have the same x_{src}.

dim	weight	dimension name
1	–	number of reflections having the same parameter name x_{src}
2	–	number of reflections having the same (t_{src}, t_{dst})
3	–	number of macro-states from t_{src} to t_{dst}
4	–	number of *already chosen* reflections having the same x_{src}
5	–	number of *already chosen* reflections having the same (t_{src}, t_{dst})

Table 4.6: Dimensions of `prioritization(reflection,chosen)`

4.4.3 Chopping, a particular form of Slicing

Slicing permits to limit the state space exploration. This technique focuses on parts of the applications w.r.t. a *slicing criterion*. The notion of slicing has been extended to model-based languages. Various techniques are proposed in the literature [Androutsopoulos *et al.* 2013]. In LigRE, we are interested in finding paths between a source t_{src} and a destination t_{dst} on the model. Thus we use a compressed form of slicing called *chopping* [Jackson & Rollins 1994], which captures this relation.Our chopping consists in a shortest sequence of transitions (also called path) starting with t_{src} and ending on the originating node of t_{dst}. We use [Dijkstra 1959]'s algorithm for computing such paths on CTFM.

Figure 3.2 illustrates a CTFM produced by step B. If the targeted reflection is $t_{dst} = (7 - (msg)\rightarrow 17))$ and $t_{dst} = (18\rightarrow 21)$, then an example of slice for this reflection is illustrated in Figure 3.3.

4.5 Implementation

The approach is implemented as a tool LigRE containing approximately 8000 SLOC of Python3.2. Figure 4.9 represents its architecture. KameleonFuzz (Chapter 5) extends LigRE by incorporating a new fuzzer. During the control flow inference, the parse tree (approximated by a subset of the Document Object Model (DOM)) is obtained using the selenium library [Huggins *et al.*] which instruments the Google Chrome browser to parse HTTP replies. During the taint flow inference, requests are performed directly to the web application. During the fuzzing, LigRE drives the application in the source via the prefix slice ; it then parameterizes the suffix slice for a fuzzer (w3af [Riancho 2011] or KameleonFuzz Chapter 5).

	Reflection										
id	t_{src}	x_{src}	t_{dst}	dim_1	dim_2	dim_3	dim_4	dim_5	*sum*	*chos.*
a	7 → 33	message2	7 → 33	1	1	0	0	0	-2	0
b	7 → 17	msg	18 → 21	1	1	1	0	0	-3	0
c	33 → 9	action	33 → 9	5	1	0	0	0	-6	0
d	18 → 21	action	21 → 9	5	1	0	0	0	-6	0

(initial state)

id	t_{src}	x_{src}	t_{dst}	dim_1	dim_2	dim_3	dim_4	dim_5	*sum*	*chos.*
a	7 → 33	message2	7 → 33	1	1	0	**1**	0	**-3**	**1**
b	7 → 17	msg	18 → 21	1	1	1	0	0	-3	0
c	33 → 9	action	33 → 9	5	1	0	0	0	-6	0
d	18 → 21	action	21 → 9	5	1	0	0	0	-6	0

(after the first iteration, *a* has been chosen)

id	t_{src}	x_{src}	t_{dst}	dim_1	dim_2	dim_3	dim_4	dim_5	*sum*	*chos.*
b	7 → 17	msg	18 → 21	1	1	1	**1**	0	**-4**	**1**
c	33 → 9	action	33 → 9	5	1	0	0	0	-6	0
d	18 → 21	action	21 → 9	5	1	0	0	0	-6	0
a	7 → 33	message2	7 → 33	1	1	0	1	0	-3	1

(after the second iteration, *b* has been chosen)

id	t_{src}	x_{src}	t_{dst}	dim_1	dim_2	dim_3	dim_4	dim_5	*sum*	*chos.*
c	33 → 9	action	33 → 9	5	1	0	**1**	0	**-7**	0
d	18 → 21	action	21 → 9	5	1	0	**1**	0	**-7**	**1**
a	7 → 33	message2	7 → 33	1	1	0	1	0	-3	1
b	7 → 17	msg	18 → 21	1	1	1	1	0	-4	1

(after the third iteration, *d* has been chosen, it could have been *c* also, see lines 21-26 in Algorithm 4.8)

Table 4.7: Prioritization of Reflections

In order to configure LigRE for an application, the tester has to write a `config.xml` file (an extract of such a file is illustrated in Listing A.2 in page 142) which contains informations about the interface (e.g., Domain Name System (DNS) Fully Qualified Domain Name (FQDN), Transmission Control Protocol (TCP) port, baseHREF, etc.), a link to the tester written reset script, the stopping condition, the nonces in the web application, and eventually some pruning patterns if the tester wants the efforts to be concentrated in specific parts of the application. The tester can also adapt the weight of the dimensions in our heuristics, altough it should not be necessary to them for applications similar to the ones on which we experimented.

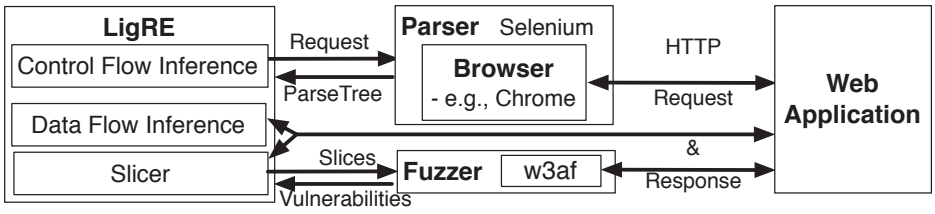


Figure 4.9: Architecture of LigRE

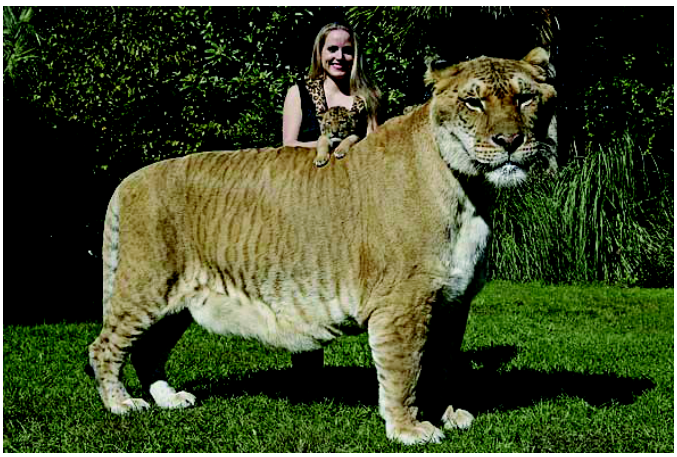


Figure 4.10: The Liger Hercules (10 feet long and 922 pounds weight), the LigRE Logo

4.6 Related Work

4.6.1 Control Flow Inference (CFM)

Based on [Angluin 1987]'s L*, [Shahbaz & Groz 2009] designed an algorithm for iteratively inferring the control flow of an I/O system. [Cho *et al.* 2010] infer a botnet protocol by adding a prediction heuristic to [Shahbaz & Groz 2009]. [Hossen *et al.* 2013] automatically generate test drivers for non-Ajax web applications.

[Doupé *et al.* 2012] showed that improving control flow inference increases vulnerability detection. LigRE shares similarities with their macro-state-aware-crawler. Differences lay in the heuristics, the introduction of confidence, contradictions, backtracking, and taint flow inference. [Doupé *et al.* 2012] run experiments on a local cloud, whereas we run ours on a laptop.

[Dessiatnikoff *et al.* 2011] cluster pages according a specially crafted distance for SQL injections [Dessiatnikoff *et al.* 2011]. [Marchetto *et al.* 2012a] dynamically infer the control flow of Ajax web applications [Marchetto *et al.* 2012a]. They wrote abstraction functions for common Ajax primitives.

[Tonella *et al.* 2012] use genetic algorithm for finding the right balance between over and under-approximations of CFM[Tonella *et al.* 2012].

LigRE does not make use of L* (because of the NDV, the macro-states which leads to enormous state machines) and is driven by heuristics. It clusters pages according to the notion of macro-state. The current implementation supports non-Ajax applications or Ajax applications which downgrade gracefully.

4.6.2 Taint Flow Inference

W3af [Riancho 2011] and XSSAuditor [Bates *et al.* 2010](Chrome XSS filter) assume the fuzzed input value to be reflected without modification, and thus rely on exact string matching. This may lead to false negatives when input values are transformed [Heiderich *et al.* 2010, Duchène *et al.* 2013b]. Skipfish generates three variants for a spiderlink, and assumes there is a taint flow if the response varies [Zalewski & Heinen 2009, Dessiatnikoff *et al.* 2011, Doupé *et al.* 2012]. This may lead to false positives, if the scanner is not aware of a macro-state change. [Sun *et al.* 2009] compute a string edit distance [Levenshtein 1966]. [Sekar 2009] proposed a filtering algorithm inspired from bioinformatics for improving the efficiency of [Levenshtein 1966]'s distance. LigRE relies on a filter-tolerant substring matching of a minimal length, and computes the edit distance on a smaller output. LigRE relies on the fuzzer test verdict.

4.6.3 Control and Taint Flow Inference (CTFM)

[Caselden *et al.* 2013] use similar models, named Hybrid Control Flow Graph (HI-CFG) on basic blocs, to automatically generate exploits for memory corruption vulnerabilities in binary programs with a grey-box test context. Netzob infers protocols implementations using L*, and enhance it with taint flows w.r.t. equivalence, size, or repetition relations. Its test driver, abstraction, and concretization functions are written by an analyst [Bossert & Guihéry 2013]. With PRISMA, [Krueger *et al.* 2012b] infer control and taint flow Markov models of botnet protocols from traffic captures. LigRE targets XSS, a command injection vulnerability, in web applications with a black-box test context, and produces CTFM to drive a fuzzer.

4.6.4 Search for Parts of the Inputs where to Focus the Testing

[Haller *et al.* 2013], [Rawat & Mounier 2012], [DeMott *et al.* 2012a], and [Bekrar *et al.* 2012, Bekrar 2013b] statically search for potential sinks and then dynamically generate inputs targeting those potential sinks.

[Stock *et al.* 2013, Duchène *et al.* 2013a] dynamically propagate white-box taint flows to prioritise DOM-XSS tests.

[Cadar *et al.* 2008b] uses symbolic execution for generating inputs s.t. each branch of an application is activated at least once by one input.

[Grégoire 2013] fuzzes third party code, and then generate inputs for applications which integrate such code (e.g., Acrobat Reader).

[Mulliner & Miller 2009] fuzzed the iOS SMS service by sending messages through the baseband and modifying them before they reach the iOS service.

4.6.5 Conclusion

LigRE automatically partially reverse-engineers web applications as a control and taint flow model. It prioritizes model slices to guide the scope of the fuzzing.

Heuristics drive LigRE. Empirical experiments show that LigRE detects more XSS than open source and control flow aware scanners (see Section 6.2).

In addition of being an input for human penetration testers, the obtained models can be the first step for automated vulnerability detection: e.g., if provided to a model checker or a fuzzer. For instance, our evolutionary smart fuzzer KameleonFuzz [Duchène *et al.* 2012, Duchène *et al.* 2013b] can use such models, and improves the fuzzing step of LigRE to detect more complex filtered XSS.

We observed there are two main reasons for false negatives: first the fuzzers neither adapt to the reflection context nor to the server sanitizers, and second they have an imprecise test verdict. We address those issues in Chapter 5.

CHAPTER 5

Evolutionary Fuzzing for Black-Box XSS Detection

If no mistake have you made, yet losing you are ...
a different game you should play.

[Yoda 2001]

Fuzzing is normally limited to finding obvious symptoms like crashes, because it's rare to be able to tell correct behavior from incorrect behavior when the input is generated randomly.

[Ruderman 2014]

5.1 Introduction

5.1.1 Context

XSS detection is a problem involving control+taint flows, and input sanitization. In presence of even basic sanitizers, many scanners have difficulties in creating appropriate inputs, and thus produce false negatives. In Chapter 4, we addressed the automatic reverse-engineering of control+taint flow models. In the current chapter, we focus on how to generate malicious inputs targeting the potential sinks. We address the following problems of Section 2.5.1 (page 33): *XSS.3.2 How to fuzz inputs? How to act on specific parts of those inputs? XSS.3.3 How to prioritize inputs fuzzing? Which potential sinks should we test first? XSS.2.2 Can we exploit a potential sink?*

In order to address these issues, we propose KameleonFuzz, a fuzzer which mimics a human attacker by evolving and prioritizing the most promising malicious inputs and taint flows obtained from LigRE. We incorporate in KameleonFuzz a test verdict that relies on existing browser parsing and double taint inference.

5.1.2 The KameleonFuzz Approach

KameleonFuzz is a black-box fuzzer which targets Type-1 (reflected) and Type-2 (stored) XSS (see Definition 8) and can generate exploits targeting the discovered

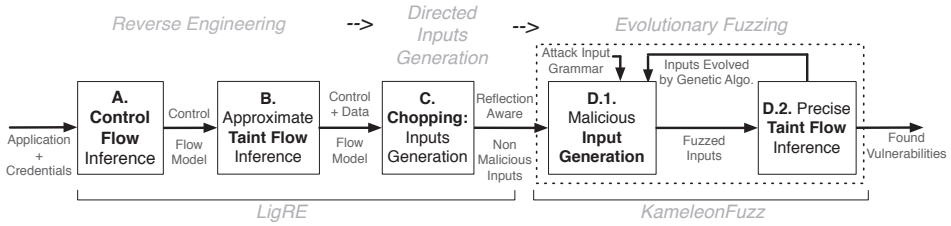


Figure 5.1: High Level Approach Overview

XSS. As illustrated in Figure 5.1, our approach consists of learning the model of the application and generating malicious inputs. We reuse the LigRE components *A, B, C* from Chapter 4. The main contribution of this chapter is KameleonFuzz which encompasses the blocks *D.1* (malicious input generation) and *D.2* (precise taint flow inference).

A Genetic Algorithm (GA), parameterized by an *Attack Input Grammar (AIG)*, evolves individuals (malicious inputs). The AIG reduces the search space and mimics the behavior of a human attacker by constraining the mutation and crossover operators which generate next generation inputs. We define a *fitness function* that favors most suitable inputs for XSS attacks. Since server sanitizers may alter the observed value at the reflection point O_{dst}, a naive substring match may not infer the taint precisely enough, which could lead to false negatives. To overcome such limitations, we perform a double taint inference. The GA iteratively evolves the best individuals of the current generation, according to their fitness score, and recombines them to produce the next generation of individuals.

5.2 Evolutionary XSS Fuzzing

The fuzzing (step D in Figure 5.1) generates a *population* of *individuals* (Genetic Algorithm (GA) terminology). An individual is an input (sequence of HTTP requests) generated by LigRE (line 7 of Algorithm 5.1) in which KameleonFuzz generates a fuzzed value x_{src} according to an Attack Input Grammar (AIG) for the reflected parameter (see line 15 of Algorithm 5.1). Inputs are concretized, sent to the application, and the corresponding outputs are recorded. Then a precise taint inference between the fuzzed value and the browser parse tree is performed in line 18. Each individual which did not find a vulnerability (test verdict of line 19 evaluates to false) is evolved via the mutation and crossover operators (Section 5.2.6, line 25 and 28 of Algorithm 5.1) w.r.t. the AIG (Section 5.2.1) and according to the fitness score (Section 5.2.5, line 22 of Algorithm 5.1).

Algorithm 5.1: Genetic Algorithm (GA) pseudo-code

```
#IN: ctfm, attack grammar, webapp, config
#OUT: vulns

# First Generation: Individuals as Input Sequences
for l in range(1,config.popul_size):
    # a reflection is choosen, and a slice produced from the CTFM
    popul[l] = Individual(ctfm.prio_get_slice(l))

vulns=[]
# Evolve the population
while(not(stopCondition())):
    for indiv in popul:
        webapp.reset()
        # generate a fuzzer value
        x_src = attack_grammar.generate(indiv.reflection_ctx)
        input_sequence = indiv.inputs.concretize(x_src)
        o = webapp.send(input_sequence)
        taint = precise_taint_infer(x_src,o,parser)
        if(verdict(taint,patterns)):
            vulns += [input_sequence]
            popul[indiv] = Individual(ctfm.prio_get_slice(len(vulns)+
                len(population)))
        else:
            indiv.fitness_compute(x_src,o,taint,M)

    children = crossover(popul.fittest([0..math.ceil(config.cross*len(
        popul))]),attack_grammar)
    for c in children:
        if(random(0,1) <= config.mutationRate):
            c.mutate(attack_grammar)
    popul_new = children
    for l in range(len(children),len(popul)):
        popul_new[l] = popul.fittest(l-len(children])

    popul = popul_new
```

5.2.1 Attack Input Grammar (AIG)

Traditional fuzzing for memory corruption consists in the application of anomaly operators on a set of bits (e.g., expanding a string, setting an integer to INT32_MAX, etc.). This does not work when fuzzing for Web Command Injec-

tion, as first the risk of memory corruption is low on web applications, and secondly when searching for XSS, the reflection must fit a certain output structure (i.e., reflection context, Definition 13). Thus, in order to constrain the search space (i.e., avoid to search in the complete space Σ^*), we use an Attack Input Grammar (AIG) for generating fuzzed values. It represents parameter values an attacker would attempt to the application. As compared to a list of payloads as in w3af and Skipfish, an AIG can generate more values, and is easier to maintain thanks to its hierarchical structure. This AIG also constrains mutation and crossover operators (lines 13, 25, 22 of Algorithm 5.1).

The knowledge used to build an AIG consists of the HTML grammar [W3C 2012b], reflection contexts (Definition 13), string transformations in the case of context change [Weinberger *et al.* 2011a], known attacks vectors [RSnake 2007, Heyes *et al.* 2012].

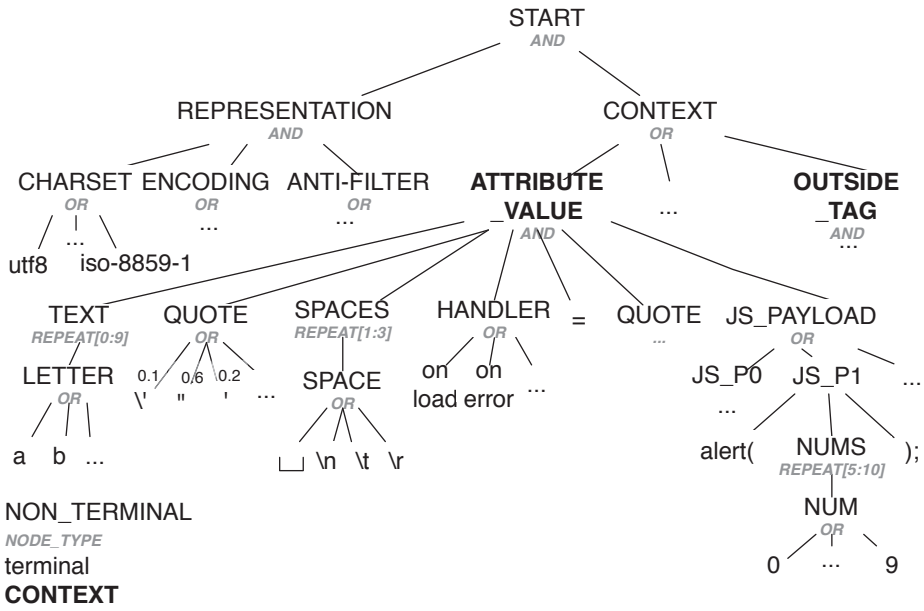


Figure 5.2: Structure of an Attack Input Grammar (AIG) (extract)

We only give a taste of how to build an AIG, as it is yet manually written and its automatic generation is a research direction. Figure 5.2 illustrates its structure. The first production rule consists of representation and context information. Example of contexts (Definition 13) include (`<input value="␣"/>`) and outside a tag (`<h1>␣`). The representation consists of encoding, charset, and special string transformation functions that we name anti-filter (e.g., PHP addslashes[PHP]). In our experiments it was sufficient to use UTF-8 encoding. However, variable length encoding such as UTF-7 [Goldsmith & Davis 1997], Shift JIS [Microsoft b], etc. may be of interest when the webpage does not specify any encoding to use.

We assume the availability of a representative set of vulnerable web applica-

tions (different from the tested applications) and corresponding XSS exploits. For each reflection context, the analyst writes a generalization of the XSS exploits in the form of production rules with terminals and non-terminals.

We represent an AIG in an Extended Backus–Naur Form [Scowen 1993] with bounded number of repetitions. We construct an AIG as an acyclic grammar. Thus it unfolds to a finite number of possibilities. Listing 5.1 contains an excerpt of an AIG we used during our experiments. The fuzzed value in Figure 5.3 was generated using this grammar.

```
1  START = REPRESENTATION CONTEXT
2  REPRESENTATION = CHARSET ENCODING ANTI_FILTER
3  CHARSET = ( "utf8" | "iso-8859-1" | ... )
4  ENCODING = ( "plain" | "base64_encode" |  ... )
5  ANTI_FILTER = ( "identity" | "php_addslashes" | ... )
6  CONTEXT = ( ATTRIBUTE_VALUE | OUTSIDE_TAG | ... )
7  ATTRIBUTE_VALUE = TEXT QUOTE SPACES HANDLER "=" QUOTE
       JS_PAYLOAD QUOTE
8  HANDLER = ( "onload" | "onerror" | ... )
9  JS_PAYLOAD = ( JS_P0 | JS_P1 | ... )
10 JS_P1 = "alert(" NUMS ")"
11 NUMS = [5:10](NUM)
12 NUM = ("0" | "1" | "2" | ... | "9")
13 QUOTE = ("'" | "\"" | "" | "\\'" | ...)
14 SPACES = [1:3](SPACE)
15 SPACE = (" " | "\n" | "\t" | "\r")
16 TEXT = [0:9](LETTER)
17 LETTER = ("a" | "b" | ...)
```

Listing 5.1: Attack Input Grammar (AIG) (excerpt)

Generating a fuzzed value consists in performing a stepwise expansion [Holler *et al.* 2012] through the production rules of an AIG and, if applicable, performing choices. Producing the corresponding string from a fuzzed value consists in concatenating the strings obtained by a depth-first exploration of the context subtree, representing this string in a given charset, applying the anti-filter function, and applying an encoding function. For instance, the string that results from the fuzzed value of Figure 5.3 is `WUkp'␣\t onload='alert(94478)`, on which the `identity` function is applied as an anti-filter, and with no encoding change (node `plain`), and the resulting string in `UTF-8` charset.

In Algorithm 5.2, we illustrate the algorithm to transduce a production tree to a concrete fuzzed value.

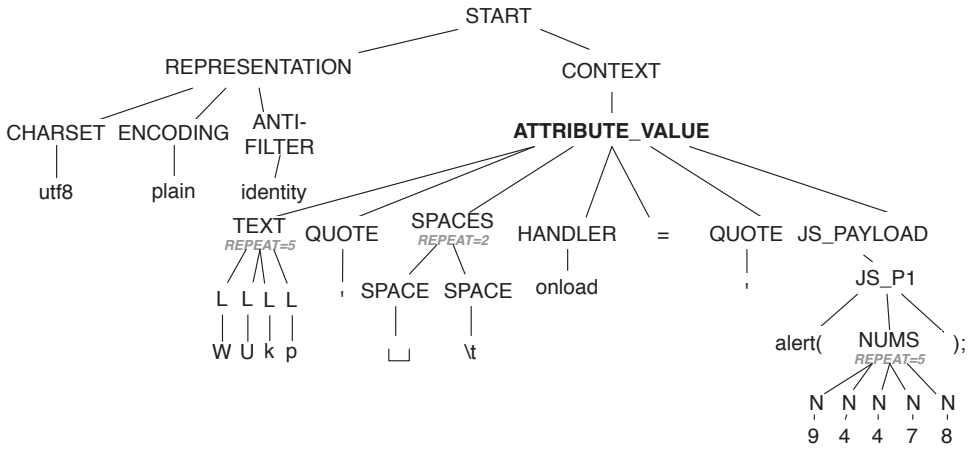


Figure 5.3: The Production Tree of a Fuzzed Value

Algorithm 5.2: AIG: From Production Tree to Concrete Fuzzed Value

```
#IN: indiv_prod_tree
#OUT: x_src (as a string)

def from_aig_word_to_string(indiv_prod_tree):
    x_src=""
    ipt = indiv_prod_tree
    x_src = DFS_aggregate(ipt.context)

    # anti-filter
    if(ipt.representation.antifilter=="identity"):
        pass
    elif(ipt.representation.antifilter=="addslashes"):
        x_src = addslashes(x_src)
    # ...

    # encoding
    if(ipt.encoding=="plain"):
        pass
    elif(ipt.encoding=="b64_encode"):
        x_src = base64_encode(x_src)
    #...

    # charset
    if(ipt.charset=="utf8"):
        pass
    elif(ipt.charset=="EUC-CN"):
        x_src = ltchinese.conversion.python_to_euc(x_src)
    #...

    return x_src
```

Algorithm 5.3: AIG: From Production Tree to Concrete Fuzzed Value (cont.)

```
def DFS_aggregate(node):
    n = len(node.children)
    if(n==0):
        str = node.value
    else:
        str = ""
        for k in range(0,n):
            str += DFS_aggregate(node.children[k])
    return str
```

In our experiments, we used the same AIG for the tested web applications. Due to a minor limitation of the current implementation, we sometimes pruned some production rules, for the search space to be narrowed, and thus the fuzzing to be faster (this can be automated easily, as it only consists of selecting the production rules for a given reflection context). We think that one unique AIG can be used when searching for Type-1 and Type-2 XSS and assuming a specific set of filters and of reflection contexts (Definition 13).

5.2.2 Individual

An individual is an input sequence targeting a specific reflection. It is composed of an input sequence as a walk in a LigRE chopped model (Section 4.4.3), and of a fuzzed value x_{src} generated from an AIG. This input encompasses the originating transition t_{src} of a taint flow, and the transition where to observe the reflection t_{dst}. We define an individual in Definition 15.

Definition 15 Individual

Let M be a CTFM (Definition 14), let $(x_{src}, t_{src}, t_{dst}, o_{dst})$ be a reflection (Definition 5) from the transition t_{src} for the value x_{src} of the parameter $name(x_{src})$.

An individual $I = (i_0, \ldots, i_n)$ is an input sequence s.t. :

- $\exists\ j, k \in [0..n]$, $j \leq k$ and t_{src} is activated by i_j and t_{dst} is activated by i_k
- the value of the input parameter $name(x_{src})$, sent as part of i_j in t_{src}, is produced by the AIG.

5.2.3 Precise Taint Flow Inference (D.2)

When fuzzing, server side sanitizers may filter fuzzed input parameter values. Thus we have to infer the taint again, and cannot only rely on results from the taint annotation in step B (Section 4.3). Moreover, we want to answer the questions

XSS.2.2 Can we exploit a potential sink? and *XSS.3.3 How to prioritize inputs fuzzing?*. So we need to track the taint up to the nodes of browser parse tree. Thus we perform this *precise* taint flow inference.

The precise taint flow inference permits obtaining information about the context of a reflection. This later will serve for computing test verdict, and as an input for the fitness function.

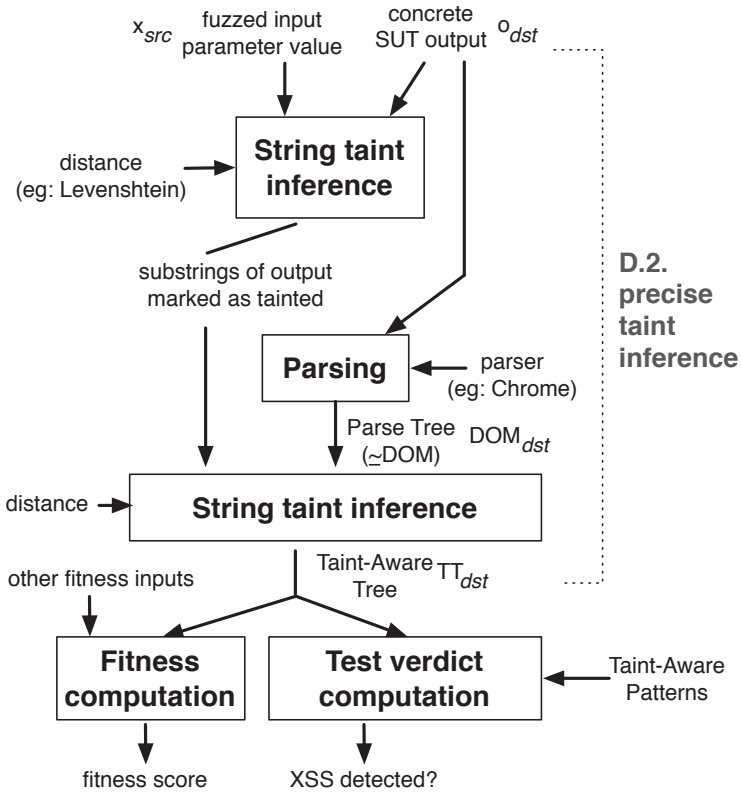


Figure 5.4: Precise Taint Inference ($x_{src} \rightarrow o_{dst} \rightarrow TT_{dst}$)

The flow for producing a Taint Aware Tree (TAT) TT_{dst} is shown in Figure 5.4. We illustrate a TAT in Figure 5.6 and define it in Definition 16. First, a string to string taint-inference algorithm (e.g., with the [Levenshtein 1966] edit distance) is applied between the fuzzed value x_{src} and the output o_{dst} in which it is reflected. This first step results in Figure 5.5. In parallel, a parser (e.g., from Google Chrome) evaluates the application output o_{dst} and produces a parse tree DOM_{dst} (e.g., a Document Object Model (DOM)). Then the taint is inferred between each tainted substring of o_{dst} and each node of DOM_{dst} to produce a TAT TT_{dst} (see Figure 5.6), as follows.

```
<input name="message2"
value="WUkp" onload="alert(94478)" />
```

Figure 5.5: Tainted Substrings of the output o_{dst}

For each node of an output parse tree DOM_{dst}, we compute a string distance between each tainted substring and the node textual value. Then we only keep the lowest distance score. If this score is lower than a tester defined threshold, then this node is marked as tainted. This taint condition may be slightly relaxed in the case of a cluster of neighbors nodes has a distance "close to the threshold". The inferred TAT TT_{dst} (e.g., Figure 5.6) is an input for the fitness function and test verdict.

The data flow from the sending of a fuzzed input parameter value to its reflection within a parse tree node involves at least two transformations (if omitting the transformations due to the encoding): a filter / sanitizer at server side, and the parsing by the browser of the concrete output o_{dst}. During the server side processing, the filter / sanitizer will induce string transformations. During the parsing, the browser may induce transformations [Weinberger *et al.* 2011a, Heiderich *et al.* 2013]. Intuitively, performing this two-steps taint inference process should increase our rate of true positive and decrease our rate of true negative, as compared to a direct string to parse tree inference.

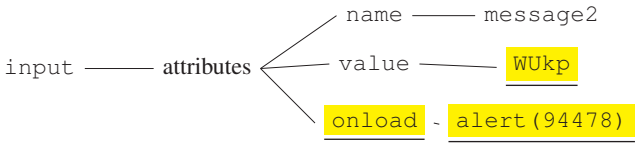


Figure 5.6: A Taint-Aware Tree (TAT) TT_{dst} (extract). The payload is a message box that displays 94478 (harmless).

It is important to note that, instead of writing our own parser, as done in [Sekar 2009], we rely on an *real-world parser*. This has two advantages. First, we are flexible with respect to the parser (e.g., for XSS: Chrome, Firefox, IE ; for other vulnerabilities such as SQL injections, we could rely on a SQL parser). Secondly, we are certain about the real-world applicability of the detected vulnerabilities. This contrasts with writing a homemade parser which may introduce false negative or false positive. However, we are aware that this potentially increases the number of cases to be tested (number of applications $\times$ number of browsers $\times$ number of versions), but the effort in searching for an XSS for a specific browser and version in a given application can be weighted depending on the number of users using that precise browser and version (which can directly relate to the allocation of testing resources and to a risk analysis).

Definition 16 A κ, μ Taint Aware Tree (TAT)

Let M be a CTFM (Definition 14). Let $(x_{src}, t_{src}, t_{dst}, o_{dst})$ be a δ-reflection in M (Definition 5).

Let $\kappa \in \mathbb{N}$ and $\mu \in [\![0..1]\!]$. A κ, μ Taint Aware Tree (TAT) consists of:

- $\Omega(o_{dst}, G)$, the parse tree of the word o_{dst} w.r.t. the grammar G.
- $d : \Sigma^{*2} \rightarrow [\![0..1]\!]$, a string distance function.
- $Z_\delta(x_{src}, o_{dst})$, the set of δ-tainted substrings in o_{dst} by x_{src}.
- a taint function $\Psi_{d,\kappa,\mu} : Z \times \Omega \rightarrow \{true, false\}$ s.t. for each tainted substring $z \in Z$ and each parse tree node $\omega \in \Omega$,

$$\Psi_{d,\kappa,\mu}(z, \omega) = \begin{cases} True & \text{if:} \\ & \quad - \text{ the set of } \kappa\text{-tainted substrings in } z \text{ by } \omega.value \text{ is not empty} \\ & \quad - \text{ OR } d(\omega.value, z) \leq \mu \\ False & \text{otherwise} \end{cases}$$

5.2.4 Test Verdict

The test verdict answers the question "Did this individual trigger an XSS vulnerability?". The TAT TT_{dst} (Figure 5.6) is matched against a set of *taint-aware tree patterns* (TAP) (e.g., Figure 5.8).

If at least one pattern matches, then the individual is an XSS exploit (i.e., the test verdict will output "yes, vulnerability detected"). Our TAP are stable w.r.t. the tested applications: we use the same TAP for all of them. A TAP is a tree containing regular expressions on its nodes. Those regular expressions may contain strings(e.g., `script`), taint markers, repetition operators($+, \star$), or the match-all character(`.`). The tester can provide its own TAP. We incorporate in KameleonFuzz default TAP for detecting successful XSS exploits. Those all violate the syntactic confinement of tainted values.

Potential Vulnerability (Non Syntactic Confinement) If the TAP illustrated in Figure 5.7 matches the TAT TT_{dst}, then there is a non syntactic confinement of a tainted value. This exhibits a *potential vulnerability*.

Figure 5.7: The generic TAP detecting non syntactic confinement of a tainted value

weight	id	dimension
+ + +	1	successfully injected character classes
+ + +	2	tainted nodes in TT_{dst}
++	3	singularity
++	4	transitions from source x_{src} to reflection o_{dst}
++	5	new page discovered
++	6	new macro-state discovered
+	7	unexpected page seen
+	8	page correctly formed w.r.t. output grammar
+	9	unique nodes from the start node

Table 5.1: Dimensions of the `fitness` function

Exploitability A second step is to match the TAT TT_{dst} with XSS specific TAP. In order to write TAP, we observed the reflection contexts (see Definition 13 in Section 4.3) and tainted parse tree nodes of outputs in various XSS attacks. Most of them attempt to craft a handler in order to trigger code execution (e.g., JavaScript) [Heyes *et al.* 2012]. From this set of attack vector, we generalize minimal tainted parse trees, which are the TAP. We illustrate an example of TAP in Figure 5.8. The second TAP in that figure matches the TAT represented in Figure 5.6. In Appendix B, we provide a detailed list of the TAP included in KameleonFuzz.

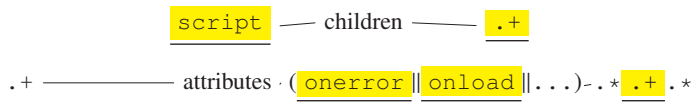


Figure 5.8: Two Taint-Aware tree Patterns (TAP), represented in a Linear Syntax (resp. a tainted script tag content and a tainted event handler attribute)

5.2.5 Fitness

The fitness function assesses "how close" is an individual to finding an XSS vulnerability. The higher its value, the more likely the GA evolution process will pick the genes of this individual for creating the next generation. The inputs of the fitness function are the individual I, the concrete output o_{dst} in which the fuzzed value x_{src}, sent in the transition x_{src}, is reflected, $T_{dst} = taint(parse(o_{dst}), x_{src})$ the taint-aware parse tree, and the application model M. The fitness dimensions are related to properties we observed between the fuzzed value and the reflection in the case of successful XSS attacks. Those dimensions are listed in Table 5.1.

Those dimensions model several intuitions that a human penetration tester may have. The most significant ones are:

- 1**: Percentage of Successfully Injected Character Classes**. Characters

that compose leaves of individual fuzzed value tree (see Figure 5.3) are categorized into classes depending on their meaning in the grammar (e.g., C_1 ={<,>} C_2 ={",'}, C_3 ={\n,\r,\t}, C_4 ={;,:}, etc.). This metric expresses the "injection power" for the considered reflection.

- 2**: Number of Tainted Nodes in** TT_{dst}. Whereas injecting several character classes is important, it is however not a sufficient condition for an attacker to exert control on several parse tree nodes. Successful XSS injections are generally characterised by at least two neighbours tainted nodes (one which is supposed to confine the reflection, and the other(s) that contain the payload and a trigger for that payload). Thus, if an attacker is able to reflect on several nodes, we expect that it increases its chances to exploit a potential vulnerability.

- 3**: Singularity of an individual w.r.t. its current generation**. A problem of GA is overspecialization that will limit the explored space and keep finding the same bugs [DeMott *et al.* 2007]. To avoid this pitfall, we compute "how singular" an individual is from its current generation. This dimension uses the source transition x_{src}, the fuzzed value x_{src}, and the reflection contexts (i.e., the destination transition o_{dst} and the tainted nodes in TT_{dst}, see Definition 13).

- 4: The higher the **Number of Transitions between the source transition** x_{src} **and its Reflection** o_{dst}, the more difficult it is to detect that vulnerability, because it expands the search tree.

- 5: a **New Page discovered** (5) **or 6: a new Macro-State** (6) **discovered**: increase application coverage.

5.2.6 Mutation and Crossover Operators

A probability distribution decides whether an individual will be mutated or not. When a mutation will happen, an operator is applied either on the *fuzzed value* or on the *input sequence*. We list the implemented fuzzed value mutation operators in Table 5.2. We fuzzed while aiming at one reflection at a time.

The *fuzzed value* mutation operator works on the production tree of the fuzzed value x_{src} (see Figure 5.3). The amplitude of the mutation is a decreasing function of the fitness score: if an individual has a high fitness score, the mutation will target nodes in the production tree that are close to leafs. Similarly, in the case of a low fitness score, the operator is more likely to mutate nodes close to the root. Figure 5.9 illustrates an example of application of the fuzzed value operator.

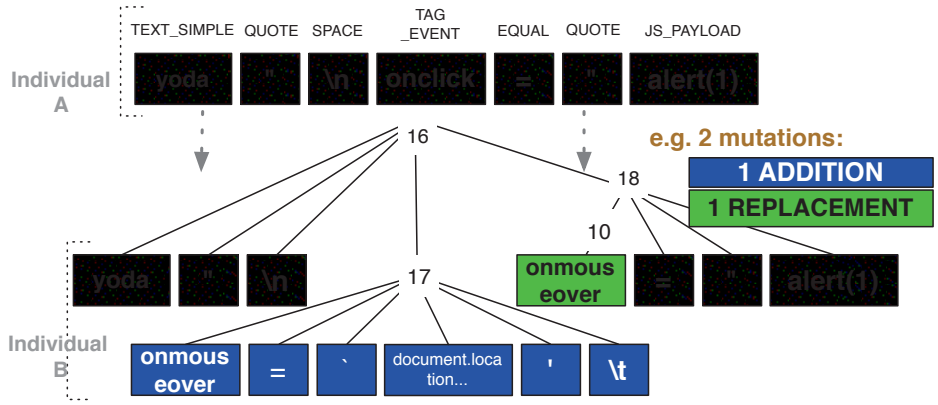


Figure 5.9: An Example of Application of a Mutation Operator on a Fuzzed Value

The *input sequence* mutation operator works on the whole sequence I. It consists in either taking another path in the model from the source $x_{src}$to the destination o_{dst}, or targeting a different reflection.

Name	Param_0_name	param_0_values
fuzzed parameter value mutation	# of sub-tree to mutate	$[\![0 \dots 2]\!]$
path mutation	max length of new input sequence	$\mathbb{N}^+$ (tester defined)

Table 5.2: The Mutation Operators

The crossover operator works at the *fuzzed value* level, i.e., on the production tree. Its inputs are two individuals of high fitness scores. It produces two children. When a crossover operator is applied on two parents which share at least one production rule at a sufficiently deep level, we exchange at most two pairs of sub-trees between the parents, in accordance with an AIG. Figure 5.10 illustrates an application of the crossover operator on the fuzzed values of parents A and B. In this figure, we only represent one of two children: the child $AB1$ contains the `TEXT`, `QUOTE`, and `SPACE` production sub-trees of A, the subtree 17 of B, the the subtree 10 of B, and the subtrees `EQUAL`, `QUOTE` and `JS_PAYLOAD` of A.

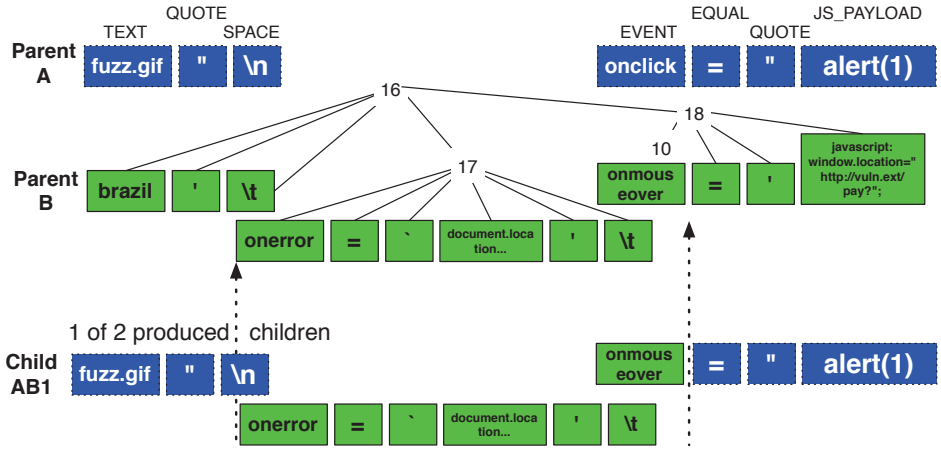


Figure 5.10: Crossover of Two Individuals *A* and *B* in KameleonFuzz (only one of two children is shown)

Name	Param_0_name	param_0_values
sub-tree exchange	# of sub-tree exchange	$[\![0 \dots 2]\!]$

Table 5.3: The Crossover Operator

5.2.7 Stopping Condition

The stopping condition is a boolean function which evaluates to true when the tester wants the fuzzing to be stopped. This function receives the number of distinct found XSS vulnerabilities, the number of founds XSS exploits, the number of submitted fuzzed inputs, the duration of fuzzing, and the number of generations evolved.

5.3 Implementation

5.3.1 Technical Details

KameleonFuzz is a python3 program which targets Type-1 and 2 XSS. It is composed of 4500 lines of code. As shown in Figure 5.11, we instrument Google Chrome [Google] with the Selenium library [Huggins *et al.*]. It includes LigRE (8.000 lines of code) (Section 4.5), as control+taint flow model inference tool and slicer. The tester has to provide an attack grammar, and stopping conditions.

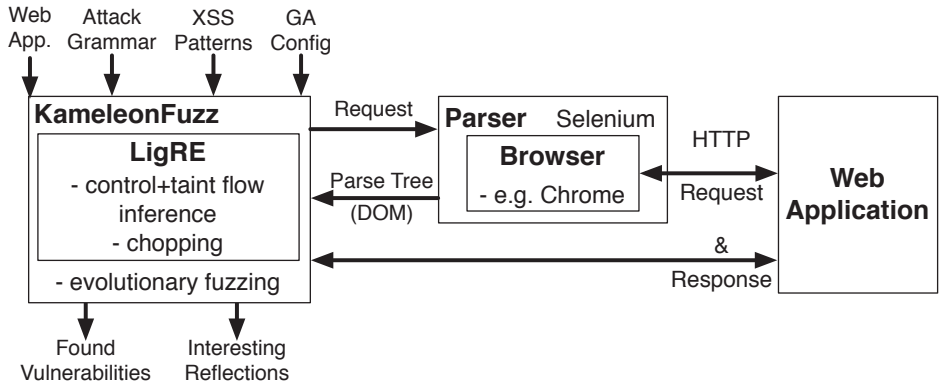


Figure 5.11: Architecture of KameleonFuzz

Figure 5.12: Pascal, the KameleonFuzz Logo

5.3.2 A Potentially Iterative Process

In practise, the tester defines its stopping conditions. At the end of the test campaign, if no interesting results were found, the tester can increase the testing resources (e.g., increasing the number of generations, the size of the population, etc.). In such a situation, our current implementation may repeat some tests which were made during the first campaign. This can be avoided by storing the results of the first campaign. This is a limitation of the implementation, not of the approach.

5.4 Related Work

5.4.1 XSS Test Verdict in a Black-Box Approach

Confinement Based Approaches assume that malicious inputs break the structure at a given level (lexical or syntactical). As in [Sekar 2009], we rely on non-syntactical confinement and we use detection policies that are both syntax and taint aware. A key difference is that [Sekar 2009] wrote his own parser to propagate the taint, whereas we use the parser of a browser (e.g., Google Chrome). Thus we infer the taint twice (see Figure 5.4). By doing so, we are sure about the real-world applicability of the found XSS exploits, and our implementation is flexible w.r.t. the browser. [Su & Wassermann 2006] relies on non-lexical confinement as a sufficient fault detection measure, which is more efficient than [Sekar 2009], but requires a correctly formed output (which is not an always valid assumption on HTML webpages [Heiderich *et al.* 2010]) and is prone to false negatives.

Regular-Expressions Based Approaches assume that the fuzzed value is reflected "as such" in the application output i.e., that the sanitizer is the identity function. In the case of sanitizers this may lead to false negatives [Riancho 2011]. Moreover, most do not consider the reflection context, which can lead to false positive. IE8 [Ross 2008] and NoScript [Maone 2006] rely on regular expressions on fuzzed values. XSSAuditor (Chrome XSS filter) performs exact string matching with JavaScriptDOM nodes [Bates *et al.* 2010].

String Distance Based Approaches Sun[Sun *et al.* 2009] detects self-replicating XSS worms by computing a string distance between DOM nodes and requests performed at run-time by the browser.

IE8 [Ross 2008] and Chrome XSSAuditor [Bates *et al.* 2010] filters only work on Type-1 XSS. Whereas NoScript [Maone 2006] is able to block some Type-2 XSS, but is only available as a Firefox plugin.

5.4.2 Learning and Security Testing

In its basic form, **fuzzing** is an undirected black-box active testing technique [Barton *et al.* 1989]. [Zalewski 2011a, Valotta 2013, Holler *et al.* 2012, Ruderman 2007] mainly target memory corruption vulnerabilities. [Stock *et al.* 2013]'s recent work fuzzes and detects Type-0 XSS in a white-box test context. [Heiderich *et al.* 2013] detects in black-box Mutation based Cross Site Scripting (m-XSS) caused by browser parser quirks. LigRE+KameleonFuzz is a black-box fuzzer which targets Type-1 and 2 XSS (Definition 8).

Genetic Algorithm (GA) for black-box security testing has been applied to evolve malwares [Noreen *et al.* 2009] and attacker scripts [Budynek *et al.* 2005].

[DeMott *et al.* 2007, Rawat & Mounier 2010, Bekrar *et al.* 2012] target memory corruption vulnerabilities in a grey-box test context. Their fitness function contains the number of executed basic blocks and the singularity of inputs. [DeMott *et al.* 2007] performs random 1-point crossover and 2-points mutation. [Rawat & Mounier 2010, Bekrar *et al.* 2012] perform offset aware mutations. *KameleonFuzz* is the first application of GA to the problem of black-box XSS search. Its fitness dimensions model the intuition of human security penetration testers.

An Attack Grammar (AIG) produces fuzzed values for XSS as a composition of tokens. [Wang *et al.* 2010, Tripp *et al.* 2013] and KameleonFuzz share this view. In their recent work, [Tripp *et al.* 2013] prune a grammar based on the test history to efficiently determine a valid XSS attack vector for a reflection. It would be interesting to compare KameleonFuzz to their approach, and to combine both. [Wang *et al.* 2010] use a hidden Markov model to build a grammar from XSS vectors. [Kals *et al.* 2006] uses attack vectors from a very large manually written library, without specific criterion.

Learning for Security Testing Radamsa targets memory corruption vulnerabilities: it infers a grammar from known inputs then fuzzes to create new inputs [Pietikäinen *et al.* 2011]. [Shu & Lee 2007] passively infer a model from network traces, and actively fuzz inputs.

For command injection vulnerabilities (XSS, SQL injection, ...), [Dessiatnikoff *et al.* 2011] cluster pages according a specially crafted distance for SQL injections. [Sotirov 2008] iterates between reverse-engineering of XSS filters, local fuzzing, and remote fuzzing. [Doupé *et al.* 2012] showed that inferring macro-state aware control flow models increases vulnerability detection capabilities.

CHAPTER 6

Experiments for XSS Detection

> You bruteforce all the moves in chess,are you playing? Mutate the code that plays chess, you are learning and playing. The seed is the code.
>
> [Heyes 2013]

> One development team cannot fail 1000 times to develop a secure application, but 1000 teams can fail one time.
>
> [Ruff 2013a]

> I enjoy making things, breaking things, and making things that break things.
>
> [Moore 2013]

> Making the world a better place... One crash a time!
>
> [Takanen 2012]

We evaluate our approach by applying our tools on different Web Applications, listed in Table 6.1. We separately evaluate the LigRE inference (Section 6.2) and the KameleonFuzz evolutionary fuzzer (Section 6.3) components. We discuss the limitations of our tools in Section 6.4.

6.1 Evaluation methodology

We selected seven **web applications** of various complexity (Table 6.1). The criteria for choices are various (different server side languages: JSP, Python, PHP; have shown to contain at least one XSS each; some are used at industrial scale). KameleonFuzz detected at least one true XSS in all of them.

P0wnMe v0.3 is an intentionally vulnerable web application for evaluating black-box XSS scanners. It contains XSS of various complexity (transitions, filters, reflection structure).

WebGoat v5.4 is an intentionally vulnerable web application for educating developers and testers. Its multiple XSS lessons range from message book to human resources.

Application	Description	Version	Plugins
P0wnMe	Intentionally Vulnerable	0.3	
WebGoat		5.4	
Gruyere		1.0	
WordPress	Blog	3.2.1	Count-Per-Day 3.2.3
Elgg	Social Network	1.8.13	
phpBB	Forum	2.0	
e-Health	Medical	04/16/2013	

Table 6.1: Tested Web Applications

Gruyere v1.0 is an intentionally vulnerable web application for educating developers and testers. Users can update their profile, post and modify "snippets" and view public ones.

Elgg v1.8.13 is a social network platform used by universities, governments. Users can post messages, create groups, update their profile. An XSS exists since several versions.

WordPress v3 is a blogging system: the blogger can create posts and tune parameters. Visitors can post comments, and search. The count-per-day plugin is known to contain XSS.

PhpBB v2 is a forum platform. We include this version, as it is notorious for containing several XSS[Bau *et al.* 2010].

e-Health 04/16/2013 is an extract of a medical platform used by patients and practitioners, developed by a company.

XSS Uniqueness an XSS is uniquely characterized by its source transition I_{src}, its parameter name, its destination transition O_{dst} and the tainted nodes in the parse tree $T(P(O_{dst}), I_{src})$. Hence if a fuzzed value is reflected twice in O_{dst}, e.g., in two different nodes in the parse tree, and for each node, the scanner generated an exploitation sequence, then we count two distinct XSS. In our experiments, the only time we had to distinguish two XSS using the nodes in the parse tree was in the Gruyere application.

Experimental Platform We run the scanners on a Mac OS X 10.7.5 platform with a 64 Bit Intel Quad-Core i7 at 2.66GHz processor, and 4GB of RAM DDR3 at 1067MHz.

6.2 LigRE evaluation

We aim at determining if control plus taint flow model aware XSS fuzzing is efficient enough to search for vulnerabilities in typical web applications. We also aim at comparing the fault detection capability of our prototype implementation LigRE against existing state of the art black-box vulnerability scanners. Relevant

metrics include the number of distinct true XSS discovered, and the number only found by a given scanner. To measure the efficiency of the scanners, we compare the number of sent requests and of found XSS. In our experiments, LigRE detected XSS missed by other scanners, and most of the XSS found by those.

We consider the following open-source black-box XSS scanners to compare with our approach: Wapiti, w3af and SkipFish. They all infer the control flow and fuzz. Their configuration i available in [Duchène 2013c]. In addition to LigRE with all its components (A,B,C and D in Figure 4.1), we also include LigRE with only A(control flow inference) and D(w3af). We denote them as $LigRE_{ABC+D}$ and $LigRE_{A+D}$. In both setups, D denotes the w3af fuzzer.

RQ1. (Fault Revealing): *Does control plus taint flow aware fuzzing find more true vulnerabilities than other scanners?*

For each scanner and application, we sequentially configure the scanner, reset the application, set a random seed to the scanner, run it against the application, and retrieve the results. We repeat this process five times, using different seeds. If possible, scanners are configured s.t. they only target XSS. We configure them with the same information (e.g., credentials). When a scanner does not handle it, we perform two sub-runs: one with the cookie of a logged-on user and one without.

We adjust parameters for the runs to last at most five hours. Beyond this duration, we stop the scanner and manually analyze the results. The number of detected XSS is the union of distinct true XSS found during the different runs. An XSS is uniquely characterized by its source transition t_{src}, its fuzzed parameter name x_{src}, its destination output t_{dst} and the structure in which the value of x_{src} is reflected. For all scanners, we manually verify XSS. During our experiments, no scanner reported false positive XSS (Skipfish reported other false positives).

Application	Inferred Taint-Flows	True XSS Detected	Nodes	Transitions
P0wnMe	28	2	13	51
WebGoat	134	4	20	80
Gruyere	23	3	30	130
WordPress	52	2	15	129
Elgg	59	1	49	214
PhpBB	213	4	63	279
e-Health	12	5	15	33

Table 6.2: $LigRE_{ABC+D}$ (D=w3af) detection capabilities on the tested applications

Table 6.2 contains the numbers of annotated reflections, found XSS, inferred nodes and transitions. This illustrates the practicality of LigRE to infer the control and taint flow models of the evaluated applications. The number of nodes and transitions, may correspond to a partial application coverage.

Figure 6.1 represents the number of detected true XSS vulnerabilities for the

considered scanners and applications. $LigRE_{ABC+D}$ (D=w3af) detected the highest number of vulnerabilities for every application, and several vulnerabilities not detected by other scanners. These results confirm [Doupé *et al.* 2012]'s experiments: improving the control flow inference($LigRE_{A+D}$) increases the vulnerability detection capabilities, as compared to non macro-state-aware scanners (Wapiti, w3af(D), Skipfish). Moreover, comparing $LigRE_{A+D}$ and $LigRE_{ABC+D}$ shows that taint flow inference(B) and slices for flow aware fuzzing(C) also increase XSS detection capabilities. We notice that $LigRE_{ABC+D}$ founds vulnerabilities missed by other scanners, including $LigRE_{A+D}$ (D=w3af): see the non-dotted part of Figure 6.1.

Most scanners achieve limited coverage due to their partial handling of basic forms, their inability to track the macro-state (beyond the classic logged in/out, and assuming the tester provides values). At times, they send requests regardless of the available links. The aggressive behavior of Skipfish is sometimes positive (e.g., in Gruyere, it found one XSS missed by others on 404 pages), sometimes not (e.g., in Wordpress, it submitted 150 times a form without detecting any XSS). As the control flow for the targeted reflection is quite simple, LigRE detected the XSS in Wordpress mainly because it sent fewer fuzzed values than SkipFish. For Elgg, both Skipfish and w3af loop between pages because they only consider the URL and not the page model.

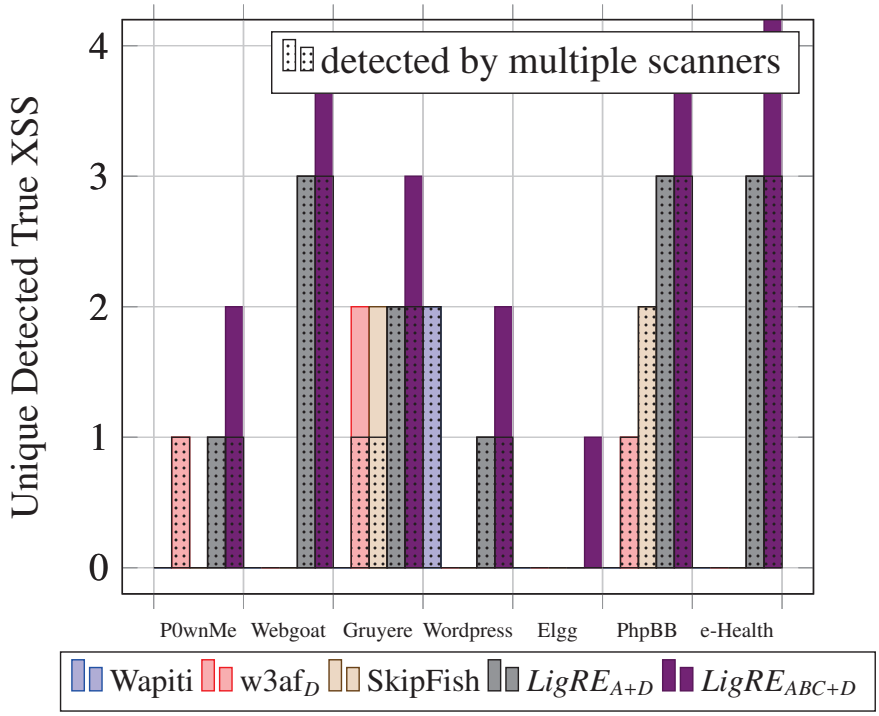


Figure 6.1: XSS Detection Capabilities of Black-Box Scanners

On considered applications, a control plus taint flow directed fuzzing increases XSS revealing capabilities.

RQ2. (Efficiency): *How efficient are the scanners in terms of vulnerability detection capabilities per number of tests?*

We set up a proxy between the scanner and the web application, and configure this proxy to limit the number of requests. We iteratively increase this limit, run the scanner, and retrieve the number of found and distinct true XSS. We manually verify them. We run such a process five times per scanner, web application, and limit. For each number of requests, for each scanner, we sum the number of unique true XSS detected for all applications. The results are illustrated in Figure 6.2.

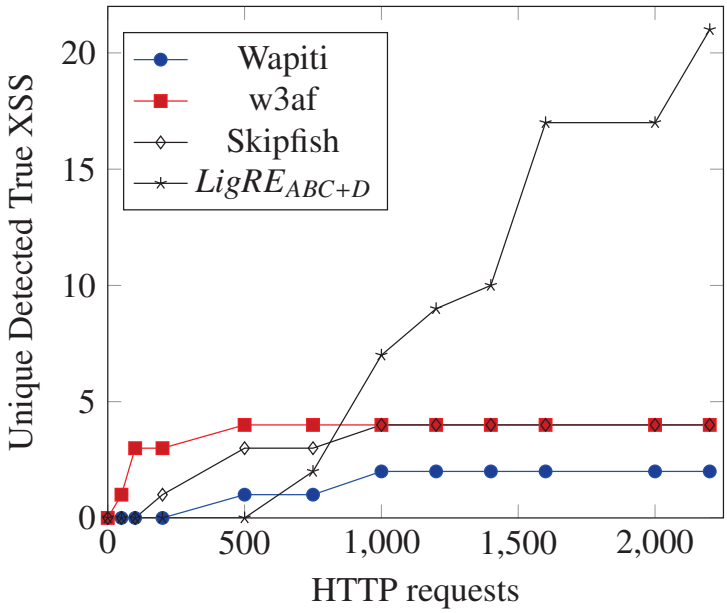


Figure 6.2: XSS Detection Efficiency of Black-Box Scanners

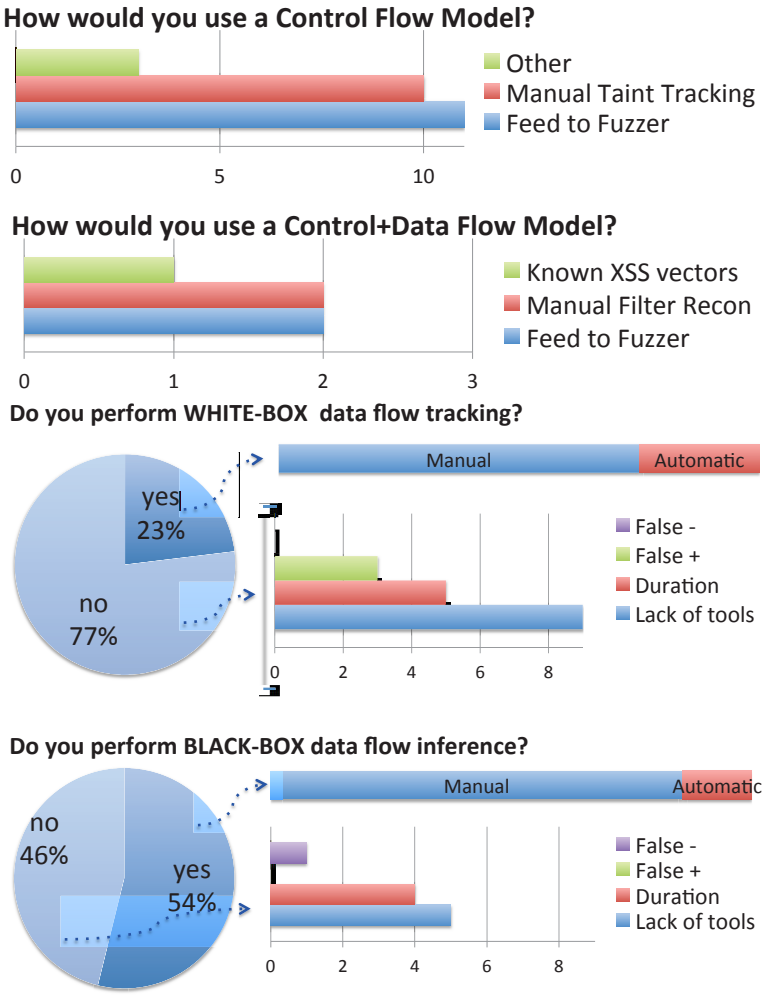
Below approximately 850 HTTP requests, w3af is the most efficient scanner. Thus we hypothesize that in applications with few macro-states, assuming it is able to navigate correctly, which in our experiments mainly happened in P0wnMe and Gruyere, then w3af is more efficient than other scanners at finding non filtered XSS.

Moreover, the LigRE proof-of-concept spends a significative number of requests in taint flow inference (from 75 to 93%). There is room for improvement. An industrial implementation should consider additional heuristics to prune sequences: e.g., with a notion of achieved coverage of n long sub-sequences.

Taint flow inference is the main barrier to entry of LigRE. If acceptable, LigRE had the highest detection capabilities. Otherwise, traditional scanners are of interest.

RQ3. (Current Use by Testers): *What is the current use of CTFM by testers?*

We conducted two surveys for evaluating the current level of use of CTFM by penetration testers[Duchène *et al.* 2013c], and how they obtained them. Figure 6.3 synthesize relevant knowledge.



Those who make use of CFM rely on a manual crawling approach, using Burp [Stuttard 2007] as a proxy, and manually draw CFM. However, since considered web fuzzers only accept a list of urls and an authentication context, they would achieve a low transition coverage.

Taint Flow Tracking 77% of testers do not perform white-box taint flow tracking, mainly because they think that not enough tools are available. 50% of those find this manual work tedious. Those who perform it rely on dynamic exact string matching. 54% of testers perform black-box taint flow inference. Most of them do it manually. 57% of those find it tedious. Taint flow tracking aims at determining the exact composing of filters, in order to produce fuzzed inputs to bypass those filters. Performing it manually is time consuming, and limited to human expertise. In white-box, it may require knowledge of various server languages. Whereas in black-box, the ability to interpret few client side languages is enough.

Even skilled penetration testers largely rely on manual taint flow tracking. Since such work is time consuming, and prone to false negatives, there is a need for tools producing hybrid control plus taint flow models.

6.3 KameleonFuzz evaluation

We evaluate our prototype implementation of KameleonFuzz against black-box open source XSS scanners, in terms of detection capabilities (RQ4) and detection efficiency (RQ5). In our experiments, KameleonFuzz detected most of the XSS detected by other scanners, several XSS missed by other scanners, and 3 previously unknown XSS.

We considered four **black-box XSS scanners** to compare with KameleonFuzz: Wapiti, w3af, Skipfish and LigRE+w3af. Appendix A contains the configuration we used during the experiments. It is important to note that only LigRE and KameleonFuzz are macro-state aware.

RQ4. (Fault Revealing): *Does evolutionary fuzzing find more true vulnerabilities than other scanners?*

To answer this question, we consider the number of true positives, the number of false positives, and the overlap of true positives. For the first two metrics, we compare all tools, whereas for the overlap, we compare LigRE+KameleonFuzz against the others (Wapiti, w3af, skipfish, LigRE+w3af). True positives are the number of XSS found by a scanner that actually are attacks, thus the higher, the better. If a scanner produces false positives, a tester will loose time, thus the lower the better. The overlap indicates vulnerabilities detected by several scanners. We denote as T_A the True XSS vulnerabilities found by the scanner A. We define the

Application	Potential Reflections	# gen. to detect the found XSS	Transitions $start \rightarrow O_{dst}$			True Positive (TP)	False Positive (FP)
			1st	2nd	3rd		
P0wnMe	37	3	4	6	7	**3**	0
WebGoat	134	2	6	7	7	**6**	0
Gruyere	23	4	2	2	7	**4**	0
WordPress	52	2	2	2	5	**4**	0
Elgg	59	1	6			**1**	0
PhpBB	213	4	5	5	6	**6**	0
e-Health	12	1	4	4	4	**8**	0

Table 6.3: KameleonFuzz detection capabilities on the considered applications

overlap as:

$$overlap(A, B) = \frac{T_A \cap T_B}{T_A \cup T_B}$$

A low overlap indicates that scanners are complementary. We also consider the vulnerabilities only detected by one scanner:

$$only\_by(A, B) = \frac{T_A}{T_A \cup T_B} - overlap(A, B)$$

A low only_by indicates that a given scanner does not find many XSS that the other missed.

For each scanner and application, we sequentially configure the scanner, reset the application, set a random seed to the scanner, run the scanner against the application, and retrieve the results. We repeat this process five times, using different seeds. Parameters have been adjusted so that each run lasts at most five hours. Beyond this period, we stop the scanner and analyze the produced results. The number of found vulnerabilities is the union of distinct true vulnerabilities found during the different runs. If possible, scanners are configured so that they only target XSS. We configure the scanners with the same information (e.g., authentication credentials). When a scanner does not handle this information correctly, we perform two sub-runs: one with the cookie of a logged-on user, and one without. Since all scanners, except LigRE and KameleonFuzz, are not macro-state aware we configure them to exclude requests that would irreversibly change the macro-state (e.g., logout when an authentication token is provided).

The **practicality of LigRE+KameleonFuzz** is illustrated in Table 6.3. This figure reports the number of potential reflections, found vulnerabilities, and generations to find all detected vulnerabilities during the fuzzing. The three columns in the middle report the length of created XSS exploits for the closest vulnerabilities from the start node.

True and False XSS Positives. We manually verify the XSS for each scanner. During our experiments, no scanner found a false positive XSS (Skipfish had other false positives). Figure 6.4 lists the results of the black-box scanners against each

application. In our experiments, KameleonFuzz detected the highest number of XSS, and several XSS missed by others. The union of the distinct true XSS found by the scanners is 35. LigRE+w3af finds $\frac{23}{35}$ = 65.7% of the known true XSS, whereas LigRE+KameleonFuzz finds $\frac{32}{35}$ = 91.4%. KameleonFuzz improves XSS detection capabilities.

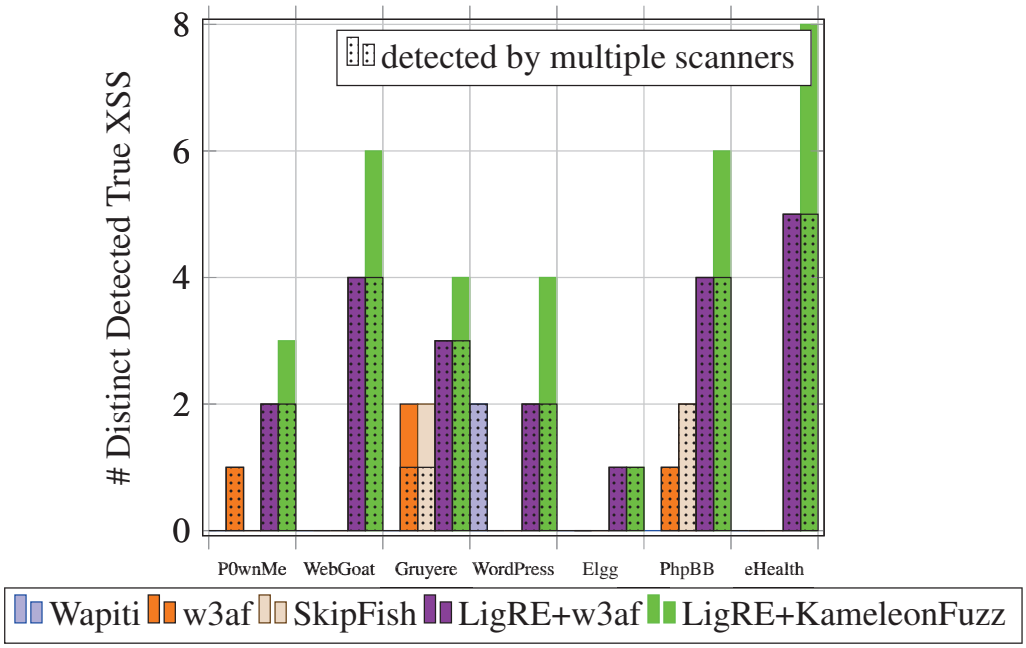


Figure 6.4: Detection Capabilities of Black-Box XSS Scanners

The overlap and only_by of true XSS found by LigRE+KameleonFuzz (KF) against other scanners are illustrated on Figure 6.5. KameleonFuzz finds the majority of known true XSS. W3af and Skipfish find the remaining ones. In the Gruyere application, Skipfish and w3af each found one vulnerability missed by all other scanners, including KameleonFuzz. Those consist of a not referenced 404 page containing a type-1 XSS, and of a type-2 XSS within the pseudo field when registering. It is harder to find the latter XSS than others: the application behaves differently as inferred when the scanner registers a new user with a fuzzed pseudo. Reusing the fuzzing learned knowledge in the inference may permit KameleonFuzz to detect this XSS. Additionally, Skipfish and w3af both detected one XSS in Gruyere that other scanners missed. Thus the only_by of Skipfish and w3af is two in Figure 6.5, whereas in Figure 6.4, one XSS is detected by both of them. Inferring the control flow for navigating to non-referenced pages may increase LigRE+KameleonFuzz XSS detection capabilities. If this is not an option, the tester should use LigRE+KameleonFuzz, Skipfish, and w3af.

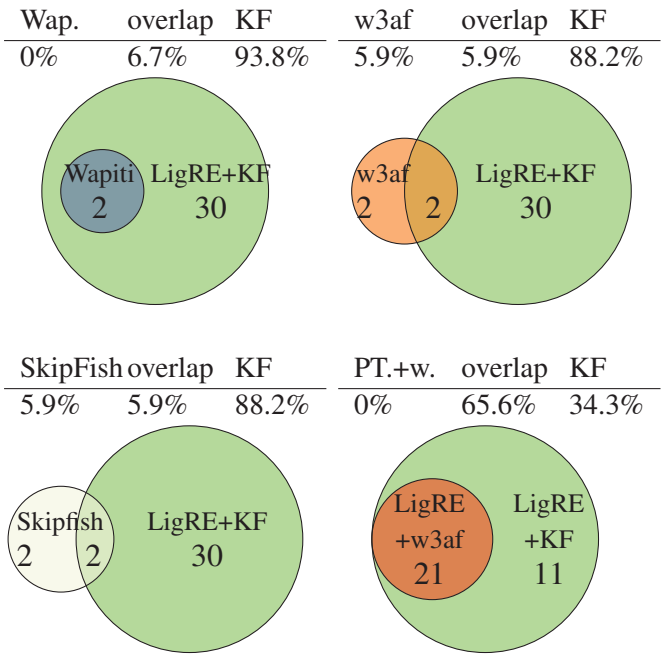


Figure 6.5: Number of True XSS found, only_by, and overlap of LigRE+KameleonFuzz and other scanners

LigRE+KameleonFuzz detects more true XSS than other scanners. It has no false positive.
KameleonFuzz increases XSS detection capabilities.
The non null only_by of w3af and Skipfish suggest they are complementary to KameleonFuzz.

RQ5. (Efficiency): *How efficient are the scanners in terms of found vulnerabilities per number of tests?*

To answer this question, it is appropriate to observe the number of detected true XSS depending of the number of HTTP requests. Thus, we set up a proxy between the scanner and the web application, and configure this proxy to limit the number of requests. We iteratively increase this limit, run the scanner, and retrieve the number of found distinct true XSS. We manually verify them. We run such a process five times per scanner, web application, and limit. For each number of requests, for each scanner, we sum the number of unique true XSS detected for all applications. The results are illustrated in Figure 6.6.

On considered applications, below approximatively 800 HTTP requests per application, w3af is the most efficient scanner. Thus we hypothesize that in applications with few macro-states, assuming it is able to

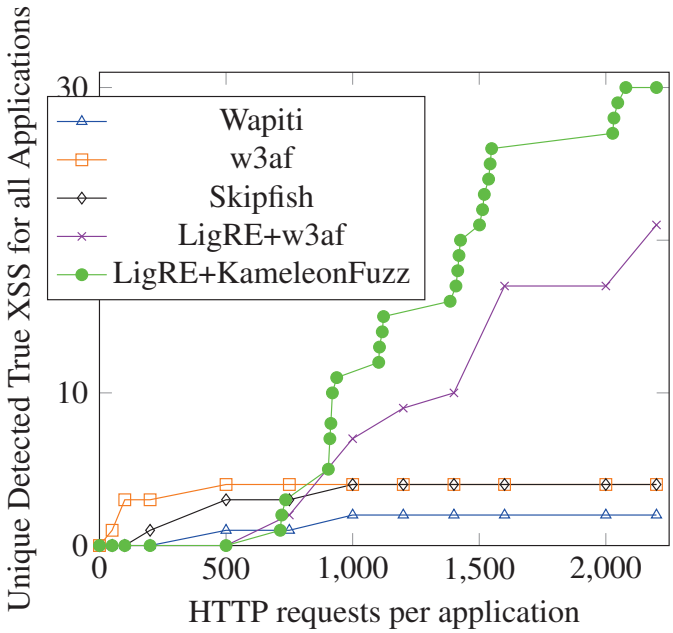


Figure 6.6: Detection Efficiency of Black-Box XSS Scanners

navigate correctly, w3af is more efficient than other scanners at finding non filtered XSS. In our experiments, mainly happened in P0wnMe and Gruyere. In applications with more macro-states, assuming the cost of control+taint flow inference is acceptable, LigRE improves vulnerability detection. Starting from 900 HTTP requests, LigRE+KameleonFuzz detects more vulnerabilities per number of requests than LigRE+w3af. For instance, after 2200 requests per application, fuzzing with KameleonFuzz detects 42.9% more XSS than fuzzing with w3af. On the LigRE+w3af and LigRE+KameleonFuzz curves, we can observe several landings, which mostly correspond to the end of the LigRE control+taint flow inference for a given application.

If the cost of LigRE inference is acceptable, then LigRE+KameleonFuzz is more efficient than LigRE+w3af.
Otherwise, w3af alone is of interest.

6.4 Discussion

6.4.1 Found 0-days XSS

During the experiments, I found the following XSS. We describe two of them in the appendix at page 155.

- CVE-2013-7297 – 1 Type-2 XSS in Elgg 1.8.13 (impact: Elgg is notably used by the Australia Governement, Wiley Publishing, the University of Florida) [Duchène 2013a]
- CVE-2014-1599 – 39 Type-1 XSS in SFR BOX NB6-MAIN-R3.3.4 (impact: ~ 5.2 Million users + possibility of changing the routing table) [Duchène 2014a]
- 1 Type-1 XSS and 2 Type-2 XSS in Siemens-Home (impact: user impersonation, customer data thief, privilege escalation) [Duchène 2014c]
- 4 Type-1 and 4 Type-2 XSS in Siemens e-Health (impact: privilege escalation) [Siemens 2014, Duchène 2014b]
- 2 Type-2 XSS in HITB CFP (impact: privilege escalation) [Duchène 2014f]
- 1 Type-1 XSS in ITEA2 Diamonds Website [Duchène 2014b]

6.4.2 Applicability to other Web Command Injections (WCI)

Even though we only experimented with Type-1 and 2 XSS vulnerabilities, we are confident that the KameleonFuzz approach can be applied to other types of WCI, with proper adaptations (e.g., attack grammar), as shown in Table 6.4. Such adaptation still do not require access to the application source code, only the ability to intercept at run-time the arguments at the observation points. Thus for command injection vulnerabilities other than Type-1 and Type-2 XSS, one may consider our approach as having a grey-box harness. Using our approach for detecting Type-0 XSS and mutation-XSS is likely to require an adaptation of the attack grammar[Heiderich *et al.* 2010, Heiderich *et al.* 2013].

Vulnerability	Output Grammar	Where to Parse?
Cross Site Scripting	HTML	HTML page
HPP Param. Pollution	HTTP	Reply Headers
PHP Code Injection	PHP	argument of `eval`
SQL Injection	SQL	arg. of `sql_query`
Shell Injection	Shell	...`exec`, `system`

Table 6.4: Command Injections: Vulnerabilities, Output Grammars, and Observation Points

6.4.3 Limitations

We classify the limitations in two categories: the limitations of our approach, and those of our implementation.

6.4.3.1 Approach Limitations

Reset We assume the ability to reset the application in its initial node, which may not always be practical (e.g., when testing a live application having users connected, we have to work on a copy). However, this does not break the black-box harness assumption: we do not require to be aware of how the macro-state is stored (e.g., database). How to relax this assumption is a research direction.

Generation of an Attack Input Grammar (AIG) Writing an AIG requires knowledge of the parameters mentioned in Section 5.2.1. This work is yet manual. The trade-off between the size of the language generated by this grammar and the fault detection capabilities is yet to be studied. A too narrow generated language (e.g., few produced fuzzed values for a given context, or very few contexts) may limit the fault detection capability, whereas a too important one may have limited efficiency. Moreover, the AIG is tied to the targeted injection subfamily (e.g., XSS, SQL injection, etc), thus the need for human input is a current limitation. There is room for research in automating this generation process [Wang *et al.* 2010].

XSS Model Hypothesis We hypothesize that an XSS is the result of only one fuzzed value. Our current approach may have false negative on XSS involving the fuzzing of at least two fuzzed values at a time[Dalili 2012]. To our knowledge, no scanner handles such cases.

6.4.3.2 Implementation Limitations

Non Deterministic Value (NDV) We assume the tester's ability to identify the NDV (e.g., anti-CSRF, viewstate, ...), or constantly changing pages [SPaCIoS 2013]. Approaches to automate their detection have been investigated [Hossen *et al.* 2011].

Dynamic Client-Side Content Our implementation does not support Ajax applications, unless they gracefully downgrade (i.e., keep their functionality while navigating via HTTP requests instead of Ajax events). [Marchetto *et al.* 2008, Amalfitano *et al.* 2008] automatically infer Ajax applications. Flash and PDF files are not yet supported. Thus on very dynamic applications such as Facebook which made intensive use of DOM transitions, our implementation would only infer and fuzz a subset of the tested application. This means that the fault detection capabilities of our implementation is limited to the subset of the application accessible by performing only HTTP requests which can be discovered in links and in forms. We did not experiment with such applications.

Encoding The precision and efficiency of the taint flow inference is dependent of the considered encoding transformations. Plain, url and base64 encodings are implemented. LigRE and KameleonFuzz can be extended to support more.

6.4.4 Threats to Validity

External Comparison We only compare to open source black-box web scanners and LigRE. We contacted several vendors of commercial products, but we did not receive a positive reply within a reasonable timeframe. Thus we were unaware to compare with commercial scanners. Those may obtain better results than the considered scanners.

Randomness Scanners make extensive use of randomness. Since some XSS are not trivial to be found, their discovery may involve randomness and duration. We tried to limit such factors by running the scanners five times with different seeds and up to five hours. The chosen duration of the experiments may impact the results.

Considered Applications Our comparison with other scanners is limited to the considered versions of scanners and applications. We cannot generalize results from those experiments. Running the scanners on other applications or scanners versions may produce different results.

KameleonFuzz Parameters KameleonFuzz contains numerous adjustable parameters e.g., probabilities that drive the mutation and crossover operators during the fuzzing. In Appendix A, we provide significant parameters and their default values. Those are chosen empirically. Because the value domain of each parameter is quite wide, and it is time consuming to run the whole test suite, it was not feasible to evaluate the combination of all parameters values and their impact. Thus, we cannot guarantee that the chosen default values achieve the best detection capabilities and efficiency.

CHAPTER 7

Other Approaches for Detecting Vulnerabilities

> Any sufficiently complicated input data is indistinguishable from bytecode, its consumer from a VM.
>
> [Bratus & Bangert 2012]

> After watching a lot of these talks I think what you have to do is ask each executive at RSA how their vision differs from modern reputation-system, brain-in-the-cloud, heuristics-based anti-viruses.
>
> [Aitel 2014]

> Sometimes, I forget that I launched calc.exe, and then I freak out when I notice it in my taskbar.
>
> [Kortchinsky 2013]

For the last three decades, in order to ensure the safety of systems, researchers proposed many techniques dedicated to the automatic search and detection of vulnerabilities. Such techniques first differ from each other depending on the test context (e.g., white-box, grey-box or black-box).

7.1 White and Grey-Box Approaches

Static Analysis techniques extract knowledge from information available without executing the application (e.g., source code). Code review is performed by automatic tools, or human security analysts. Even corporations with very large web application code base, such as Google and Facebook, still perform manual code reviews as part of their security testing processes. [Shar & Tan 2012] extracts the composition of sanitizers from the source code, and verifies if a bypass can be found for the composed sanitizer. [Aydin *et al.* 2014] extract signatures from source code to generate CTFM-like automatons, from which they generate input sequences towards achieving specific coverage criteria. Model inference from source code generate CFM, e.g., [Mihancea & Minea 2014] for

Java application. They need to balance between over and under approximation [Sankaranarayanan *et al.* 2013].

[Stock *et al.* 2013] crawls in the first reachable transitions from the start node from various web applications, propagated the taint in white-box in javascript objects and submitted XSS attack vectors from a library. Even though only the first transitions are covered, they found numerous DOM Cross Site Scripting (a.k.a. Type-0 XSS) (DOM-XSS), which shows that those web applications were not tested w.r.t. such vulnerabilities.

Grey-Box Approaches generally combine static and dynamic analysis techniques, at the assembly level. [Rawat & Mounier 2012] statically search for Buffer Overflow (BOF) sinks at the assembly level. [DeMott *et al.* 2012b, Bekrar *et al.* 2012, Bekrar *et al.* 2011] use data taint propagation at the assembly level to determine where to apply anomaly operators. Depending on the complexity and required precision of the process, the taint can be inferred in web application [Madou *et al.* 2008], or propagated precisely (e.g., [Doupé *et al.* 2013] in .Net applications).

Symbolic Execution consists in expressing branches choices w.r.t. conditions on inputs. Iteratively, the application is restarted and inputs are modified for the application to take another branch. The goal is to increase branch coverage. The SAGE fuzzer by [Godefroid *et al.* 2008] found various Memory Corruption Vulnerabilities (Memory Corruption Vulnerability (MCV)) in Microsoft Office products. Microsoft uses it daily and use their idle computing power to solve billions of constraints [Bounimova *et al.* 2013]. KLEE by [Cadar *et al.* 2008a] found many vulnerabilities in open-source projects. Those approaches assume the availability of the source code. Symbolic execution techniques have also been used for finding Web Command Injections (WCI) in web applications: [Kiezun *et al.* 2009, Wassermann *et al.* 2008, Saxena *et al.* 2010a] generate string constraints and then solve those to generate input sequences containing one fuzzed value. Symbolic execution has also been applied in a grey-box context (e.g., at binary level): [Campana 2009]. Recently, concolic execution is gaining adoption, as it permits to simplify certain constraints by using concrete values and thus reduces the search space. However, all the public work still assumes the availability of the source code [Haller *et al.* 2013, Feist *et al.* 2013].

Symbolic execution techniques are starting to be used in passive testing: [Mouttappa *et al.* 2013b].

7.2 Black-Box Approaches

Black-Box security testing distinguish between passive techniques (e.g., monitoring) and active techniques which submit inputs to the applications. Black-box techniques produce a knowledge, often in the form of a model or a grammar, from which inputs are produced.

[Mouttappa *et al.* 2013a] verifies that properties are not violated on the recorded traces. [Johns *et al.* 2008] passively monitors HTTP traffic, tokenize JS in the outputs, and detects Type-1 and Type-2 XSS with an interesting precision.

[Offutt & Abdurazik 1999] generate tests from UML specifications written by analysts, and have coverage criteria specific to UML diagrams. [Friedman *et al.* 2002] generate tests from Finite State Machine (FSM) that correspond to system specifications. Similar approaches are mentioned in [Utting & Legeard 2010]. [Lebeau *et al.* 2013] manually model the application as a UML diagram with OCL guards, from which they automatically generate security tests for web applications.

Model Inference is a reverse engineering technique that consumes an application and produces a model. [Angluin 1987] pioneered the black-box model inference with her L* algorithm. [Cho *et al.* 2010] apply [Shahbaz & Groz 2009]'s algorithm to understand the behavior of a botnet C&C and to take it down. [Cho *et al.* 2011] infer CFM of protocol server implementations using concolic execution techniques from which fuzzed inputs are generated. [Shu & Lee 2007] learns using L* models of an implementation and applies model checking techniques for searching for security properties violation (e.g., confidentiality or integrity of a given object). [Li *et al.* 2006] separately infer components models and then compose them for creating a model corresponding to the aggregation of the components. [Petrenko *et al.* 2014] infer models containing Non Deterministic Values (NDV) (e.g., models of web applications containing viewstate). [Krueger *et al.* 2012a] learns CTFM for simulating network honeypots. [Choudhary *et al.* 2013] learns Asynchronous Javascript And XML (Ajax) application models, which are significantly larger than traditional web applications model. [Mariani *et al.* 2012] incrementally learns updates of Ajax application models.

Black-Box fuzzing techniques historically target MCV [Barton *et al.* 1989]. The beautiful story of how [Barton *et al.* 1989] crashed Unix utilities because of transmission errors – due to a storm impacting his modem connectivity – explains why black-box fuzzing is also called structural testing. In the domain of black-box interpreter fuzzing [Woo *et al.* 2013] schedule and prioritize tests according to various coverage criteria. In the *LangFuzz* approach, [Holler *et al.* 2012] require a target grammar and code fragments which it mutates. LangFuzz exposed previously unreported JavaScript and PHP vulnerabilities. [Guo *et al.* 2013] infer the grammars accepted by web browsers and mutate, according to this grammar, code fragments known to have triggered a problem. [Wen *et al.* 2013] generate programs in two phases: first they generate abstract scripts, and then they replace abstract labels with concrete identifiers (e.g., variable name, function call, etc.). They target ActionScript. They exposed previously unreported vulnerabilities in the Adobe Flash ActionScript virtual machine. [Householder & Foote 2012] apply a statistical theory algorithm to the problem of selection of parameters (randomization seed, file seed, range).

Model checking searches for properties violations on states of a model. Such models can be written by a human analyst, extracted using a white-box con-

text from e.g., the source code [Balint & Minea 2011], or inferred in black-box [Shahbaz & Groz 2009] via the observed behavior at the application interfaces.

CHAPTER 8

Discussion & Conclusion

> Sure: today, it's so easy to phish users or exploit real RCE bugs, that backdooring web origins is not worth the effort. But in a not-too-distant future, that balance may shift.
>
> [Zalewski 2011c]

> Give me ten carefully chosen hackers, and within 90 days I would then be able to have this nation lay down its arms and surrender.
>
> J. Saiteerdou, FBI [Liang & Xiangsui 1999]

> Vulnerability Detection Systems: Think Cyborg, Not Robot
>
> [Heelan 2011]

8.1 Discussion

We propose several directions of research for overcoming limitations of the implementation, applying to other web command injections, and applying to other classes of vulnerabilities, such as memory corruption.

8.1.1 Influence of Various Parameters

Encoding When no charset is specified for a page which is a destination for a reflection (i.e., no content-type HTTP header, or no charset as a child of the `<head>` node, or the reflection happens before this tag), it is interesting to act on the encoding in the attack input grammar (AIG), in order to find an XSS. In our experiments we only used UTF-8 encoding, as it already permitted to find XSS in various applications. However, variable length encoding such as UTF-7 [Goldsmith & Davis 1997], Shift JIS [Microsoft b], etc.may be of interest in the aforementioned case.

Evolutionary Parameters Studying thoroughly the influence of evolutionary parameters (e.g., crossover and mutation rate, population size, fitness weights) may be of interest. The main difficulty is the significant number of combinations to test, and the required duration to test thoroughly each given combination.

8.1.2 Improving the efficiency of attack grammar

Automatic Attack Grammar Generation Few researchers addressed the problem of the automatic generation of an attack grammar. [Wang *et al.* 2010] pioneered this direction by inferring one grammar using a hidden Markov model. Ideally an interesting grammar balances between a precision sufficient to bypass filters and a narrowed search space s.t. the testing campaign for a given reflection context has an acceptable cost. [Tripp *et al.* 2013] have an interesting compromise with their advanced representation which combines filters, and reflection contexts.

Web Filters Reverse-Engineering In the process of fuzzing a reflection, knowledge can iteratively be learned regarding the filter, thus there is room for applying machine learning algorithms. For instance, if the sanitizer is on client side (e.g., as in most DOM-XSS), precise automata may be built using static analysis techniques [Saxena *et al.* 2010b], e.g., symbolic execution [Saxena *et al.* 2010a] or even concolic execution [Cho *et al.* 2011]. However, when the sanitizer is at server side, such techniques cannot be used. This research direction has been pioneered by [Sotirov 2008]. [Tripp *et al.* 2013] implicitly capture the notion of Web Filters Models in their attack grammar. If we would infer using machine learning techniques the transducer for each reflection, then we would be able to prune our attack grammar sub-tree with a great precision.

Attack Grammar Pruning A method for increasing the efficiency of Web Command Injection fuzzing consists in iteratively pruning sets of attack vectors after observing the output obtained form each fuzzed input sequence. [Tripp *et al.* 2013] exactly adopt such an approach.

8.1.3 Adapting the Approach in case of other Counter-Measures

In our problem, we hypothesize the only presence of server-side sanitizers. In this section, we list the other defensive measures which it may be interesting to address.

Client-Side Sanitizers (inside the browser) transform input parameters values based on black-listed regular expressions [nos 2006, Ross 2008] [Bates *et al.* 2010], or prevent the execution of inline scripts[1] if this script was sent in an input parameter value of the current HTTP request [Google , nos 2006]. [Ross 2013]'s jSanity is an Internet Explorer (IE) client side sanitizer based on IE9+'s parser. Its configuration is performed in a white-listing fashion (e.g., allow foo-bar ; deny foo-*).

Web Application Firewall (WAF) is a firewall performing Deep Packet Inspection (DPI) up to the HTTP layer. [qua , mod , iro 2011] are examples of WAF.

[1] In the webpage `<script> code </script>`.

They act as active defense mechanism, by detecting and removing content within the webpages. They may even parse and interpret the webpage as the an end-user browser would do. Since their parser may be different from the end-user, there is a risk of false negatives and of false positives.

Static Rewriting [Doupé *et al.* 2013]'s deDacota statically rewrites .Net web applications in their binary form to enforce code+data separation when concatenated variables or inlined JavaScript code creation has been detected.

Trusted DOM [Heiderich *et al.* 2011] a white-listing policy in which the developer indicates which DOM modification operations she allows. While this requires work not to break the functionality of applications, such a counter-measure can prevent type-0, type-1, and type-2 XSS.

HTML5 IFrame Sandbox Several XSS vulnerabilities exist due to browser API developers not enforcing a strict access control model [Heiderich *et al.* 2010], several attacks consists in a webpage embedded in an iframe to access its parent DOM. In order to limit the impact in such cases, the HTML5 IFrame Sandbox [w3c 2009] permits the parent webpage to disable capabilities of the embedded one such as script execution, form submission etc.

DOM Tags Randomization [Van Gundy & Chen 2009] randomizes the prefix of HTML tags for each request. For successfully exploiting a XSS vulnerability, an attacker would have to guess the prefix that will be picked up next. This makes the reliability of an XSS exploit very low. Randomizing the position of certain codes is a similar concept to the ASLR [Yarom 1999, Team 2012], a 2001 counter-measure for raising the cost of exploiting memory corruption vulnerabilities. Slightly differently, [Kc *et al.* 2003] randomizes the instruction set through a XOR mechanism specific to each process and run.

Content Security Policy (CSP) [w3c 2012a, Google 2013] declares a policy at server side, and enforces it at client side. The default policy prevents the execution of inline scripts, only allows `<script src="url.js" />`, and disables `eval()` like functions, and only allows the loading of "local scripts" (i.e., with the same security domain that is same server Fully Qualified Domain Name (FQDN) and TCP port) and resources. The main hypothesis behind such counter-measures is that attackers exploiting XSS first execute inline JavaScriptwhich loads a more complete library from a remote server they control. This server different of the server hosting the application under attack. CSP enforces at client side a data and code separation policy, which is defined at server side.

Unfortunately, as promising as those counter-measures may seem, their current deployment is quite low. In Table 8.1, we list possible reasons behind the low adoptions of the aforementioned XSS counter-measures.

Defense	**Barriers to adoption**	**Server Impact**	**Client Impact**
Client-Side Sanitizer	- adherence to browser version - may break applications		x
WAF	may break applications		
Trusted DOM	may break applications	x	
HTML5 IFrame Sandbox	scope very limited (e.g., not applicable for Java, Flash)	x	x
DOM randomization	- may break applications - only make exploit less reliable	x	x
CSP	- non trivial amount of adaptations [Weinberger *et al.* 2011b] - 96% of Alexa top 1000 use CSP-incompatible code patterns [Golubovic 2013]	x	
Static Rewriting	- may break applications - limited to ASP.Net applications	x	

Table 8.1: Difficulties behind the Low Adoption of XSS Counter-Measures

8.1.4 Application to other Web Command Injections (WCI)

Abstraction Level When performing automatic control flow inference for one given HTTP driver, the obtained automata are of significant size. When extending our LigRE+KameleonFuzz approach to other type of WCI such as DOM-XSS, Ajax XSS, Cross Protocol XSS, or even SQL injections, the major challenge researchers will face is likely to be the size of the obtained CFM. Depending on the abstraction, the inferred CFM of an application such as Facebook may be composed of several millions of nodes. If the abstraction is performed at a too high level, specificities of WCI taint flows may be missed ; if the abstraction is performed at a too low level, it may be unpractical to infer a complete model and thus transitions may be missed.

Automatic Detection of Cross Protocol XSS Automatically detecting Cross-Protocol XSS is a research direction. From the test context perspective, it involves two abstraction and concretization functions, thus there is the need of developing additional test drivers for handling protocols such as SMTP, FTP, URL handlers. Penetration testers may not systematically search for such vulnerabilities, thus some interesting results such as Figure C may be obtained.

Automatic Detection of Type-0/DOM XSS While [Stock *et al.* 2013] discovered numerous DOM-XSS, it is important to note that they only traversed the very first Ajax transitions from the start node. This means that deeper vulnerabilities are likely to have been overlooked. Thus the CFM should first be extended to take into account transitions of the DOM which could be fired by JavaScript events. There has been work in the inference of Ajax CFM [Marchetto *et al.* 2008, Marchetto *et al.* 2012b]. A useful model for fuzzing would be a composition of CTFM as obtained by LigRE and of CTFM extracted from taint propagation mechanism in the JavaScript virtual machine (e.g., [Vogt *et al.* 2007], Dominator Pro [Paola 2011]). For detecting Type-1 and Type-2 XSS, the same Taint Aware tree Patterns (TAP) used in KF can be used. From the test verdict perspective, the only difference lies in the matching of the taint aware parse tree (TAT) against TAP after each transition (i.e., classic HTTP or Ajax transitions). Very preliminary results obtained with students suggest that taint aware DOM-XSS grammar based evolutionary fuzzing may increase the efficiency of DOM XSS fuzzers [Duchène *et al.* 2013a].

Automatic Detection of Flash XSS Similarly to the automatic detection of DOM-XSS, the automatic detection of Flash Type-1 and Type-2 XSS involves extending the CFM models by taking into account transitions in abstract Flash machine states and reflections originating and reaching those transitions. An implementation can use tools such as FlashDOM [Murray 2010].

Application to other Web Command Injections (WCI) We believe that the LigRE+KameleonFuzz approach can be applied to various types of web command injections (WCI): e.g., SQL injection [Su & Wassermann 2006], Shell command injection [Sekar 2009], PHP interpreted code injection. This requires adaptations on the points where we observe the reflections, so such an approach would then be considered as having a grey-box test context. An implementation could hook server side APIs such as `sql_query()`, `shell_exec()`, `eval()`, etc.

Relaxing the Single Parameter Fuzzed Value Hypothesis Many fuzzers and penetration testers search for XSS using only one reflection. However, XSS may involve several tainted fuzzed parameter values. Even companies such as Google acknowledge that such scenarii are a hard combinatorial problem: for a Type-1 XSS to be catched by the Google Chrome filter, it has to be constructed with only one reflected fuzzed value and to use at least one special character in the HTML grammar(e.g., `"`). If two or more parameters are reflected, this scenario is unsupported by the filter [Nikiforakis & Barth 2011]. In order to search for two parameter XSS, the similar CTFM may be used. The selection strategy should focus on transitions exhibiting at least two reflected parameters. Such adaptations would permit the automatic detections of the vulnerabilities mentioned in page 155:

Type-2 double parameter confusion XSS in Siemens-Home [Duchène 2014c] and 1 Type-2 double parameter confusion XSS in `https://mega.co.nz`.

GUI Security Testing Black-box model inference can be applied to GUI desktop applications such as Evernote. We discovered manually a Unconstrained URL Handler (CVE-2014-1404) [Duchène 2014e]. Such a vulnerability could have been detected using CTFM which contains reflections within the arguments of `libc` functions such as `exec()`.

8.1.5 Evolutionary Black-Box Fuzzing for Memory Corruption

Our recent work [Duchène 2014f, Duchène 2013b] combines Genetic Algorithm and Anti-Random Testing for detecting memory corruption vulnerabilities in interpreter. We alternatively prune the search space using a fitness heuristic and explore other directions using anti-random testing techniques. Preliminary results suggest such an approach increases the efficiency for detecting not very deeply embedded memory corruption vulnerabilities in interpreters.

8.1.6 Using Control+Taint Flow Models for Defensive Security

From the perspective of a defender, Content Security Policy (Content Security Policy (CSP)) is a counter-measure implemented in recent browsers (IE10, Firefox, Chrome), which permits web applications developers to enforce a policy regarding cross domain inclusions and inline scripts. If defined wisely, such policies could permit browsers to prevent XSS attacks. The current barrier before to the adoption of CSP is the necessity for web application developers to write such policies (e.g., Mozilla foundation ask for human web developers to help them rewrite pages on the Mozilla website). Thus there is the need for an automatic rewriting of web application, in order to make them CSP-aware. [Golubovic 2013] proposed a first candidate solution. However, in order not to break functionalities, automatic application rewriting needs to infer precisely the control flow of the application. [Golubovic 2013] does not consider the macro-state and thus may lack precision. The precision of a combination of our LigRE approach with their automatic CSP generation is a direction of research.

8.2 Conclusion

The research we contributed aims at showing the advantages of combining techniques such as reverse-engineering and evolutionary driven fuzzing for automatically detecting web command injections vulnerabilities in a black-box test context.

Since we hypothesize a black-box test context, we send inputs, record outputs, perform a computation and send new inputs, based on the computation result. The reverse engineering step consists in inferring a control flow automaton, then

annotating it with taint flows, then producing chopped models. The control flow inference addresses the problem of navigating in the application and the macro-state awareness. The taint flow inference exhibits paths containing reflections of input parameter values. The chopping constraints the fuzzing transition search space. The fuzzing step consists in the genetic evolution of individuals. An individual is composed of a chopped model and of a fuzzed value generated by an attack grammar. Once the individual is submitted, a double taint inference up to the nodes of the browser parse tree, using string edit distance, produces a taint aware tree. We designed taint aware patterns, which assess precisely the exploitability of potential vulnerabilities, depending if they match the obtained taint aware tree. The population of individuals is evolved w.r.t. a heuristic fitness function, but also mutation and crossover operators. The evolution stops when a tester defined stopping condition is met.

We discovered 0-days XSS in widely used applications: notoriously 1 Type-2 XSS in Elgg 1.8.13 (CVE-2013-7297), 39 Type-1 XSS in SFR-DSL-box (5.2M DSL boxes, CVE-2014-1599). The main barrier of entry of our approach lays in the noise induced by the number of resets and requests, which makes it unsuitable for a military offensive security operation which is supposed to be discrete, and which may be of a too high cost for several companies. In such a case, we advocate a combination of human tester guiding a hybrid inference plus fuzzing tool. For extending this work, we suggest first to adapt the test drivers for detecting DOM-XSS and Flash XSS. Then as the second research direction, we suggest to explore how to relax the single parameter fuzzed value hypothesis, as it increases significantly the complexity of the taint inference, and of the fuzzed input values creation.

Bibliography

[Abgrall *et al.* 2012] E. Abgrall, Yves Le Traon, M. Monperrus, S. Gombault, M. Heiderich et A. Ribault. *XSS-FP: Browser Fingerprinting using HTML Parser Quirks*. arXiv preprint arXiv:1211.4812, 2012.

[Acunetix 2010] Acunetix. *Exploiting a cross-site scripting vulnerability on Facebook*, 2010. http://www.acunetix.com/websitesecurity/xss-facebook/.

[Aitel 2014] Dave Aitel. *On Phillippe Courtot's RSAC Keynote*, 2014.

[Amalfitano *et al.* 2008] D. Amalfitano, A.R. Fasolino et P. Tramontana. *Reverse engineering finite state machines from rich internet applications*. In 15th WCRE, pages 69–73. IEEE, 2008.

[Androutsopoulos *et al.* 2013] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke et Laurence Tratt. *State-Based Model Slicing: A Survey*. ACM Computing Surveys, 2013.

[Angluin 1987] Dana Angluin. *Learning regular sets from queries and counterexamples*. Information and computation, vol. 75, no. 2, pages 87–106, 1987.

[Armando *et al.* 2008] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar et Llanos Tobarra. *Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps*. In 6th FMSE, pages 1–10. ACM, 2008.

[Augsten *et al.* 2005] Nikolaus Augsten, Michael Böhlen et Johann Gamper. *Approximate matching of hierarchical data using pq-grams*. In 31st VLDB, pages 301–312, 2005.

[Avižienis *et al.* 2004] Algirdas Avižienis, Jean-Claude Laprie et Brian Randell. *Dependability and its threats: a taxonomy*. In Building the Information Society, pages 91–120. Springer, 2004.

[Aydin *et al.* 2014] Abdulbaki Aydin, Muath Alkhalaf et Tevfik Bultan. *Automated Test Generation from Vulnerability Signatures*. In ICST, 2014.

[Balint & Minea 2011] Mihai Balint et Marius Minea. *Automatic inference of model fields and their representation*. In Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs, page 9. ACM, 2011.

[Barth 2011] Adam Barth. *HTTP State Management Mechanism*, 2011. http://tools.ietf.org/search/rfc6265.

[Barton *et al.* 1989] Miller Barton, L. Fredriksen et B. So. *An empirical study of the reliability of operating system utilities*. Communications of The ACM, 1989.

[Bates *et al.* 2010] Daniel Bates, Adam Barth et Collin Jackson. *Regular expressions considered harmful in client-side XSS filters*. In WWW, pages 91–100, 2010.

[Bau *et al.* 2010] Jason Bau, Elie Bursztein, Divij Gupta et John C. Mitchell. *State of the Art: Automated BlackBox Web Application Vulnerability Testing*. In IEEE S&P, pages 332–345, 2010.

[Bau *et al.* 2012] Jason Bau, Frank Wang, Elie Bursztein, Patrick Mutchler et John C. Mitchell. *Vulnerability Factors in New Web Applications: Audit Tools, Developer Selection & Languages*. Rapport technique, Stanford, 2012.

[Bekrar *et al.* 2011] Sofia Bekrar, Chaouki Bekrar, Roland Groz et Laurent Mounier. *Finding Software Vulnerabilities by Smart Fuzzing*. In International Conference on Software Testing, Verification, and Validation, pages 427–430, 2011.

[Bekrar *et al.* 2012] Sofia Bekrar, Chaouki Bekrar, Roland Groz et Laurent Mounier. *A Taint Based Approach for Smart Fuzzing*. In SECTEST with ICST, pages 818–825, 2012.

[Bekrar 2013a] Chaouki Bekrar. *tweet*, July 2013. https://twitter.com/cBekrar/status/357865619099615233.

[Bekrar 2013b] Sofia Bekrar. *Recherche de Vulnérabilités Logicielles par Fuzzing Intelligent d'Exécutables*. Thèse, Université of Grenoble, 2013. http://www.theses.fr/s93520.

[Bosman *et al.* 2011] Erik Bosman, Asia Slowinska et Herbert Bos. *Minemu: The world's fastest taint tracker*. In Recent Advances in Intrusion Detection, pages 1–20. Springer, 2011. http://www.few.vu.nl/~herbertb/papers/minemu_raid11.pdf.

[Bossert & Guihéry 2013] Georges Bossert et Frédéric Guihéry. *Security Evaluation of Communication Protocols in Common Criteria using Netzob*. In ICCC, 2013.

[Bounimova *et al.* 2013] Ella Bounimova, Patrice Godefroid et David Molnar. *Billions and billions of constraints: Whitebox fuzz testing in production*. In Proceedings of the 2013 International Conference on Software Engineering, pages 122–131. IEEE Press, 2013.

[Bratus & Bangert 2012] Sergey Bratus et Julian Bangert. *Page Fault Liberation Army*. In CCC, 2012.

[Braun & Heiderich 2013] Frederik Braun et Mario Heiderich. *X-Frame-Options: All about Clickjacking?* 2013. https://cure53.de/xfo-clickjacking.pdf.

[Budynek *et al.* 2005] Julien Budynek, Eric Bonabeau et Ben Shargel. *Evolving computer intrusion scripts for vulnerability assessment and log analysis*. In GECCO. ACM, 2005.

[Cadar *et al.* 2008a] Cristian Cadar, Daniel Dunbar et Dawson R Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.* In OSDI, volume 8, pages 209–224, 2008.

[Cadar *et al.* 2008b] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill et Dawson R Engler. *EXE: automatically generating inputs of death*. ACM Transactions on Information and System Security (TISSEC), vol. 12, no. 2, page 10, 2008.

[Campana 2009] Gabriel Campana. *Fuzzgrind: an automatic fuzzing tool*. Hack. lu, 2009.

[Caselden *et al.* 2013] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Laszlo Szekeres, Stephen McCamant et Dawn Song. *Transformation-aware Exploit Generation using a HI-CFG*, 2013.

[Chikofsky *et al.* 1990] Elliot J Chikofsky, James H Cross*et al.* *Reverse engineering and design recovery: A taxonomy*. Software, IEEE, vol. 7, no. 1, pages 13–17, 1990.

[Cho *et al.* 2010] Chia Yuan Cho, Domagoj Babick, Eui Chul Richard Shin et Dawn Song. *Inference and analysis of formal models of botnet command and control protocols*. In ACM CCS, pages 426–439, 2010.

[Cho *et al.* 2011] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu et Dawn Song. *MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery.* In USENIX Security Symposium, 2011.

[Choudhary *et al.* 2013] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M Mirtaheri, Guy-Vincent Jourdan, Gregor v Bochmann et Iosif Viorel Onut. *Building rich internet applications models: Example of a better strategy*. In Web Engineering, pages 291–305. Springer, 2013.

[CVE-2006-4565 2006] CVE-2006-4565. *JavaScript Regular Expression Heap Corruption in Mozilla SeaMonkey*, 2006. Priit Laes, CanadianGuy, Girts Folkmanis, Catalin Patulea, https://www.mozilla.org/security/announce/2006/mfsa2006-57.html.

[CVE-2008-1380 2008] CVE-2008-1380. *Mozilla SeaMonkey JavaScript Garbage Collector Memory Corruption Vulnerability*, 2008. http://www.securityfocus.com/bid/28818/info.

[Czarnecki *et al.* 2000] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde et Todd Veldhuizen. *Generative programming and active libraries*. In Generic Programming, pages 25–39. Springer, 2000.

[Dalili 2012] Souroush Dalili. *Browsers anti-XSS methods in ASP (classic) have been defeated!*, 2012. http://soroush.secproject.com/downloadable/Browsers_Anti-XSS_methods_in_ASP_(classic)_have_been_defeated.pdf.

[DeMott *et al.* 2007] Jared D. DeMott, Richard J. Enbody et William F. Punch. *Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing*. Black Hat USA, 2007.

[DeMott *et al.* 2012a] Jared DeMott, Richard J. Enbody et William F. Punch. *Systematic bug finding and fault localization enhanced with input data tracking*. Computers & Security Research, 2012.

[DeMott *et al.* 2012b] Jared DeMott*et al.* *Systematic bug finding and fault localization enhanced with input data tracking*. Computers & Security, 2012.

[Dessiatnikoff *et al.* 2011] Anthony Dessiatnikoff, Rim Akrout, Eric Alata, Mohamed Kaaniche et Vincent Nicomette. *A clustering approach for web vulnerabilities detection*. In 17th PRDC, pages 194–203. IEEE, 2011.

[Dijkstra 1959] Edsger W Dijkstra. *A note on two problems in connexion with graphs*. Numerische mathematik, vol. 1, no. 1, pages 269–271, 1959.

[Doupé *et al.* 2012] A. Doupé, L. Cavedon, C. Kruegel et G. Vigna. *Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner*. Usenix Sec, 2012.

[Doupé *et al.* 2013] Adam Doupé, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel et Giovanni Vigna. *deDacota: toward preventing server-side XSS via automatic code and data separation*. In CCS, pages 1205–1216. ACM, 2013.

[Duchène *et al.* 2012] Fabien Duchène, Roland Groz, Sanjay Rawat et Jean-Luc Richier. *XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing*. In SECTEST with ICST, pages 815–817, 2012.

[Duchène *et al.* 2013a] Fabien Duchène, Wassim Al Safwi, Benoit Le Queau, Maxime Peyrard et Bastien Scanu. *Taint Assisted DOM-XSS Fuzzing with Dominator Pro*, 2013. http://ensiwiki.ensimag.fr/index.php/4MMSR-Network_Security-2012-2013-taint_assisted_dom_xss_fuzzing.

[Duchène *et al.* 2013b] Fabien Duchène, Sanjay Rawat, Jean-Luc Richier et Roland Groz. *Fuzzing Intelligent de XSS Type-2 Filtrés selon Darwin: KameleonFuzz*. In 11th SSTIC, pages 289–311, 2013.

[Duchène *et al.* 2013c] Fabien Duchène, Sanjay Rawat, Jean-Luc Richier et Roland Groz. *A Hesitation Step into the Black-box: Heuristic based Web Application Reverse Engineering*. In NoSuchCon, 2013.

[Duchène *et al.* 2013d] Fabien Duchène, Sanjay Rawat, Jean-Luc Richier et Roland Groz. *LigRE: Reverse-Engineering of Control and Data Flow Models for Black-Box XSS Detection*. In 20th WCRE, pages 252–261. IEEE, 2013.

[Duchène 2013a] Fabien Duchène. *CVE-2013-7297– Type-2 XSS in Elgg 1.8.13*, 2013. http://community.elgg.org/discussion/view/1200698/elgg-blog-elgg-1814-and-1719.

[Duchène 2013b] Fabien Duchène. *Fuzz in the Dark: Genetic Algorithm for Black-Box Fuzzing*. In Black-Hat, São Paulo, Brazil, 2013.

[Duchène 2013c] Fabien Duchène. *LigRE - scanners configuration*, 2013. http://car-online.fr/ligre_scan_conf.

[Duchène 2014a] Fabien Duchène. *CVE-2014-1599 – 39 Type-1 XSS in SFR BOX NB6-MAIN-R3.3.4*, 2014. http://seclists.org/bugtraq/2014/Mar/30.

[Duchène 2014b] Fabien Duchène. *0-day XSS discovered with KameleonFuzz*, 2014. http://car-online.fr/0_day_xss_kameleonfuzz.

[Duchène 2014c] Fabien Duchène. *1 Type-1 XSS and 2 Type-2 XSS in Siemens-Home*, 2014.

[Duchène 2014d] Fabien Duchène. *Cross Protocol XSS in XXXXXX*, 2014. Responsible disclosure pending.

[Duchène 2014e] Fabien Duchène. *CVE-2014-1404 – Evernote 5.4.4 (402282) – Unconstrained file:// Handler*, 2014. https://evernote.com/security/.

[Duchène 2014f] Fabien Duchène. *Harder, Better, Faster Fuzzer : Advances in Black-Box Evolutionary Fuzzing*. In Hack In The Box (HITB), Amsterdam, Netherlands, 2014.

[Duwell 2013] Ron Duwell. *Sony Fined £400,000 for the 2011 PlayStation Network Hack*, 2013. http://www.technobuffalo.com/2013/01/25/sony-fined-400000-for-the-2011-playstation-network-hack/.

[Einstein 1955] Albert Einstein. *Conversations avec Pablo Casals : souvenirs et opinions d'un musicien*, 1955.

[epsylon 2012] epsylon. *XSSer*, 2012. http://xsser.sourceforge.net/.

[Faghani & Saidi 2009] Mohammad Reza Faghani et Hossein Saidi. *Malware propagation in online social networks*. In 4th MALWARE, pages 8–14. IEEE, 2009.

[Feist *et al.* 2013] Josselin Feist, Laurent Mounier et Marie-Laure Potet. *Statically Detecting Use After Free on Binary Code*. GreHack, 2013.

[Filiol 2013a] Eric Filiol. *L'avance technologique de Vupen*, 2013. http://cvo-lab.blogspot.fr/2013/10/lavance-technologique-de-vupen.html.

[Filiol 2013b] Eric Filiol. *Sécurité IT : veut-on faire de l'argent ou vraiment protéger les gens?*, 2013. http://www.itespresso.fr/eric-filiol-esiea-securite-it-veut-on-faire-de-largent-ou-vraiment-proteger-les-gens-60220.html.

[Forrester & Miller 2000] Justin Forrester et Barton Miller. *An empirical study of the robustness of Windows NT applications using random testing*. In 4th USENIX Windows System Symposium, pages 59–68, 2000.

[Friedman *et al.* 2002] G. Friedman, A. Hartman, K. Nagin et T. Shiran. *Projected State Machine Coverage for Software Testing*. SIGSOFT Softw. Eng. Notes, vol. 27, no. 4, pages 134–143, Juillet 2002. http://doi.acm.org/10.1145/566171.566192.

[Godefroid *et al.* 2008] Patrice Godefroid, Michael Y Levin, David A Molnar*et al. Automated Whitebox Fuzz Testing*. In NDSS, volume 8, pages 151–166, 2008.

[Goldsmith & Davis 1997] D. Goldsmith et M. Davis. *UTF-7 RFC 2152*, 1997. http://tools.ietf.org/rfc/rfc2152.txt.

[Golubovic 2013] Nicolas Golubovic. *autoCSP - CSP-injecting reverse HTTP proxy*. BSc thesis, Rhur Bochum University - Google, 2013.

[Google] Google. *Chrome*. https://www.google.com/chrome/.

[Google 2013] Google. *Content Security Policy (CSP) - Chrome Extensions*, 2013. http://developer.chrome.com/extensions/contentSecurityPolicy.html.

[Grugq 2013] The Grugq. *Twitter discussion*, 2013. https://twitter.com/thegrugq/status/394136642094505985.

[Grégoire 2013] Nicolas Grégoire. *Dumb-fuzzing XSLT engines in a smart way*. In NoSuchCon, 2013.

[Guilfanov 2008] Ilfak Guilfanov. *Decompilers and beyond*. In HITB, 2008.

[Guo *et al.* 2013] Tao Guo, Xin Wang Puhan Zhang et Qiang Wei. *GramFuzz: Fuzzing Testing of Web Browsers Based on Grammar Analysis and Structural Mutation*. In ICIA, 2013.

[Haller *et al.* 2013] István Haller, Asia Slowinska, Matthias Neugschwandtner et Herbert Bos. *Dowsing for overflows: a guided fuzzer to find buffer boundary violations*. In Usenix Security, 2013.

[Hansen 2013] Robert Hansen. *Interview with a Black-Hat*, 2013. http://blog.whitehatsec.com/interview-with-a-blackhat-part-1/.

[Heelan 2011] Sean Heelan. *Vulnerability Detection Systems: Think Cyborg, Not Robot*. IEEE Security & Privacy, vol. 9, no. 3, pages 74–77, 2011.

[Heiderich *et al.* 2010] M. Heiderich, E.A.V. Nava, G. Heyes et D. Lindsay. Web application obfuscation:'-/wafs.. evasion.. filters//alert (/obfuscation/)-'. Syngress, 2010.

[Heiderich *et al.* 2011] Mario Heiderich, Tilman Frosch et Thorsten Holz. *Iceshield: Detection and mitigation of malicious websites with a frozen dom*. In Recent Advances in Intrusion Detection, pages 281–300. Springer, 2011.

[Heiderich *et al.* 2013] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius et Edward Z. Yang. *mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations*. In CCS. ACM, 2013.

[Heiderich 2012a] Mario Heiderich. *HTML5 security*, 2012. http://html5sec.org/.

[Heiderich 2012b] Mario Heiderich. *IE7 specific tag injection using %*, 2012. http://html5sec.org/#134.

[Heiderich 2013a] Mario Heiderich. *The InnerHTML Apocalypse*. In Insomni'Hack, 2013. http://www.slideshare.net/x00mario/the-innerhtml-apocalypse.

[Heiderich 2013b] Mario Heiderich. *JSMVCOMFG – To sternly look at JS MVC and Templating Frameworks*. In OWASP, editeur, WASR with AppSec EU, 2013. https://appsec.eu/wasr/.

[Heyes *et al.* 2012] Gareth Heyes, Alex Inführ, Mario Heiderich et Various. *Shazzer - Shared XSS Fuzzer*, 2012. http://shazzer.co.uk.

[Heyes 2013] Gareth Heyes. *tweet*, 2013. https://twitter.com/garethheyes/status/367045976210419712.

[Holler *et al.* 2012] C. Holler, K. Herzig et A. Zeller. *Fuzzing with Code Fragments*. In 21st Usenix Security, 2012.

[Hossen *et al.* 2011] Karim Hossen, Roland Groz et Jean-Luc Richier. *Security Vulnerabilities Detection Using Model Inference for Applications and Security Protocols*. In SECTEST with ICST, pages 534–536. IEEE, 2011.

[Hossen *et al.* 2013] Karim Hossen, Jean-Luc Richier, Catherine Oriat et Roland Groz. *Automatic Generation of Test Drivers For Model Inference of Web Applications*. In SECTEST with ICST. IEEE, 2013.

[Householder & Foote 2012] Allen D Householder et Jonathan M Foote. *Probability-Based Parameter Selection for Black-Box Fuzz Testing*. Rapport technique, Carnegie Mellon University, 2012. http://www.sei.cmu.edu/reports/12tn019.pdf.

[Huang *et al.* 2010] Lin-Shung Huang, Zack Weinberg, Chris Evans et Collin Jackson. *Protecting browsers from cross-origin CSS attacks*. In Proceedings of the 17th ACM conference on Computer and communications security, pages 619–629. ACM, 2010.

[Huggins *et al.*] Jason Huggins, Paul Hammant*et al.* *Selenium, Browser Automation Framework*. http://code.google.com/p/selenium/.

[iro 2011] *IronBee - Open Source WAF*, 2011. https://www.ironbee.com/dl/ironbee-whitepaper.pdf.

[Jackson & Rollins 1994] Daniel Jackson et Eugene J. Rollins. *A new model of program dependences for reverse engineering*. In 2nd SIGSOFT FSE, pages 2–10. ACM, 1994.

[Johns *et al.* 2008] Martin Johns, Björn Engelmann et Joachim Posegga. *Xssds: Server-side detection of cross-site scripting attacks*. In Computer Security Applications Conference, 2008. ACSAC 2008. Annual, pages 335–344. IEEE, 2008.

[Kals *et al.* 2006] Stefan Kals, Engin Kirda, Christopher Kruegel et Nenad Jovanovic. *Secubat: a web vulnerability scanner*. In Proceedings of the 15th international conference on World Wide Web, pages 247–256. ACM, 2006.

[Kc *et al.* 2003] Gaurav S Kc, Angelos D Keromytis et Vassilis Prevelakis. *Countering code-injection attacks with instruction-set randomization*. In Proceedings of the 10th ACM conference on Computer and communications security, pages 272–280. ACM, 2003.

[Kiezun *et al.* 2009] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer et Michael D Ernst. *HAMPI: a solver for string constraints*. In Proceedings of the eighteenth international symposium on Software testing and analysis, pages 105–116. ACM, 2009.

[Kortchinsky 2013] Kostya Kortchinsky. *Tweet*, 2013. https://twitter.com/crypt0ad/status/211858388932497409.

[Krebs 2012] Brian Krebs. *Yahoo Email-Stealing Exploit Fetches £700*, 2012. http://krebsonsecurity.com/2012/11/yahoo-email-stealing-exploit-fetches-700/.

[Krueger *et al.* 2012a] Tammo Krueger, Hugo Gascon, Nicole Krämer et Konrad Rieck. *Learning stateful models for network honeypots*. In Proceedings of the 5th ACM workshop on Security and artificial intelligence, pages 37–48. ACM, 2012.

[Krueger *et al.* 2012b] Tammo Krueger, Hugo Gascon, Nicole Krämer et Konrad Rieck. *Learning Stateful Models for Network Honeypots*. In AISEC. ACM, 2012.

[Kugler 2013] Robert Kugler. *PayPal.com XSS Vulnerability*, 2013. http://seclists.org/fulldisclosure/2013/May/163.

[Lebeau *et al.* 2013] Franck Lebeau, Bruno Legeard, Fabien Peureux et Alexandre Vernotte. *Model-Based Vulnerability Testing for Web Applications*. In Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pages 445–452. IEEE, 2013.

[Levenshtein 1966] VI Levenshtein. *Binary coors capable of correcting deletions, insertions, and reversals*. In Soviet Physics-Doklady, volume 10, 1966.

[Li *et al.* 2006] Keqin Li, Roland Groz et Muzammil Shahbaz. *Integration testing of components guided by incremental state machine learning*. In Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings, pages 59–70. IEEE, 2006.

[Liang & Xiangsui 1999] Qiao Liang et Wang Xiangsui. Unrestricted warfare. PLA Literature and Arts Publishing House Arts, 1999.

[Lin *et al.* 2009] Xiaoli Lin, Pavol Zavarsky, Ron Ruhl et Dale Lindskog. *Threat Modeling for CSRF Attacks*. In Computational Science and Engineering, 2009. CSE'09. International Conference on, volume 3, pages 486–491. IEEE, 2009.

[Livshits 2012] Benjamin Livshits. *Dynamic Taint Tracking in Managed Runtimes*. Rapport technique, Microsoft Research, 2012. https://research.microsoft.com/pubs/176596/tr.pdf.

[Luo *et al.* 2009] Weimin Luo, Jingbo Liu, Jing Liu et Chengyu Fan. *An analysis of security in social networks*. In 8th DASC, pages 648–651. IEEE, 2009.

[Madou *et al.* 2008] Matias Madou, Edward Lee, Jacob West et Brian Chess. *Watch what you write: Preventing cross-site scripting by observing program output*. In OWASP AppSec 2008 Conference (AppSecEU08), 2008.

[Magazinius *et al.* 2013] Jonas Magazinius, Billy Rios et Andrei Sabelfeld. *Polyglots: Crossing Origins by Crossing Formats*. In CCS. ACM, 2013.

[Maone 2006] Giorgio Maone. *NoScript, Firefox plug-in*, 2006. https://addons.mozilla.org/en-US/firefox/addon/noscript/.

[Marchetto *et al.* 2008] A. Marchetto, P. Tonella et F. Ricca. *State-based testing of Ajax web applications*. In ICST, pages 121–130. IEEE, 2008.

[Marchetto *et al.* 2012a] A Marchetto, P Tonella et F Ricca. *ReAjax: a reverse engineering tool for Ajax web applications*. Software, pages 33–49, 2012.

[Marchetto *et al.* 2012b] Alessandro Marchetto, Paolo Tonella et Filippo Ricca. *ReAjax: a reverse engineering tool for Ajax Web applications*. IET Software, pages 33–49, 2012.

[Mariani *et al.* 2012] Leonardo Mariani, Alessandro Marchetto, Cu D Nguyen, Paolo Tonella et Arthur Baars. *Revolution: automatic evolution of mined specifications*. In Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, pages 241–250. IEEE, 2012.

[Microsoft a] Microsoft. *CSS Expressions*. http://msdn.microsoft.com/en-us/library/ms537634(vs.85).aspx.

[Microsoft b] Microsoft. *Shift-JS Encoding (SJIS) / MS Kanji*. http://msdn.microsoft.com/en-US/goglobal/cc305152.aspx.

[Microsoft 2004] Microsoft. *ASP.Net View-State*, 2004. http://msdn.microsoft.com/en-us/library/ms972976.aspx.

[Mihancea & Minea 2014] Petru Florin Mihancea et Marius Minea. *jModex : Model Extraction for Verifying Security Properties of Web Applications*. In IEEE CSMR-WCRE 2014 Software Evolution Week (Tool track), 2014.

[Miller 2010] Charlie Miller. *Interview with Charlie Miller*, 2010. http://blogs.securiteam.com/index.php/archives/1352.

[mod] *ModSecurity - Open Source WAF*. http://www.modsecurity.org.

[Moore 2013] HD Moore. *hdm.io - about*, 2013. http://hdm.io/index.html.

[Moulu 2013] André Moulu. *Sécurité des applications Android constructeurs et réalisation de backdoors sans permission*. In SSTIC, 2013. https://www.sstic.org/2013/presentation/securite_applications_android_constructeurs_et_realisation_de_backdoors_sans_permission/.

[Mouttappa *et al.* 2013a] Pramila Mouttappa, Stephane Maag et Ana R. Cavalli. *Monitoring Based on IOSTS for Testing Functional and Security Properties: Application to an Automotive Case Study*. In COMPSAC, pages 1–10, 2013.

[Mouttappa *et al.* 2013b] Pramila Mouttappa, Stephane Maag et Ana R. Cavalli. *Using passive testing based on symbolic execution and slicing techniques: Application to the validation of communication protocols*. Computer Networks, vol. 57, no. 15, pages 2992–3008, 2013.

[Mulliner & Miller 2009] Collin Mulliner et Charlie Miller. *Fuzzing the phone in your phone*. Black Hat USA, vol. 25, 2009.

[Murray 2010] Kyle I Murray. *FlashDOM: interacting with flash content from the document object model*. In Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility, pages 311–312. ACM, 2010.

[Naraine 2010] Ryan Naraine. *Apache.org hit by targeted XSS attack, passwords compromised*, 2010.

[Newsome & Song 2005] James Newsome et Dawn Song. *Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software*. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA. Internet Society, 2005.

[Nikiforakis & Barth 2011] Nick Nikiforakis et Adam Barth. *Chromium – Issue 96616: Security: Google Chrome Anti-XSS filter circumvention*, 2011. http://code.google.com/p/chromium/issues/detail?id=96616.

[Nikiforakis & Vigna 2013] Nick Nikiforakis et Giovanni Vigna. *Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting*. In CCS. ACM, 2013.

[Nirgoldshlager 2013] Nirgoldshlager. *Stored XSS In Facebook*, 2013. http://www.breaksec.com/?p=6129.

[Noreen *et al.* 2009] Sadia Noreen, Shafaq Murtaza, M. Zubair Shafiq et Muddassar Farooq. *Evolvable malware*. In GECCO, pages 1569–1576. ACM, 2009.

[nos 2006] *NoScript : Firefox Add-On*, 2006. http://noscript.net/.

[Offutt & Abdurazik 1999] Jeff Offutt et Aynur Abdurazik. Generating tests from uml specifications. Springer, 1999.

[OWASP] OWASP. *WebGoat - the vulnerable web application*. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[OWASP 2013a] OWASP. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*, 2013. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[OWASP 2013b] OWASP. *Top Ten Project*, 2013. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[Paola 2011] Stefano Di Paola. *DOM XSS Identification and Exploitation*, 2011. http://media.hacking-lab.com/scs3/scs3_pdf/SCS3_2011_Di_Paola.pdf.

[Petrenko *et al.* 2014] Alexandre Petrenko, Keqin Li, Roland Groz, Karim Hossen et Catherine Oriat. *Inferring Approximated Models for Systems Engineering*. In 15th IEEE International Symposium on High Assurance Systems Engineering (HASE 2014), pages 249–253, Miami, Florida, USA, 2014.

[PHP] PHP. *addslashes function*. http://php.net/manual/en/function.addslashes.php.

[Pietikäinen *et al.* 2011] Pekka Pietikäinen, Aki Helin, Rauli Puuperä, Atte Kettunen, Jarmo Luomala et Juha Röning. *Security Testing of Web Browsers*. Comm. of Cloud Software, vol. 1, no. 1, Dec. 23, ISSN 2242-5403, 2011.

[qua] *QualysGuard Web Application Firewall*. https://www.qualys.com/forms/web-application-firewall/.

[Rawat & Mounier 2010] Sanjay Rawat et Laurent Mounier. *An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A Case of Aiming in Dim Light*. In EC2ND, 2010.

[Rawat & Mounier 2012] Sanjay Rawat et Laurent Mounier. *Finding Buffer Overflow Inducing Loops in Binary Executables*. In Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on, pages 177–186. IEEE, 2012.

[Riancho 2011] A Riancho. *w3af - WebApp. Attack and Audit Framework*, 2011. http://w3af.sourceforge.net.

[Ross 2008] David Ross. *IE 8 XSS Filter Implementation*, 2008. http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[Ross 2013] David Ross. *Insane in the IFRAME – The case for client-side HTML sanitization*. In AppSec EU. OWASP, 2013.

[RSnake 2007] RSnake. *XSS Cheat Sheet Esp: for filter evasion*, 2007. http://ha.ckers.org/xss.html.

[Ruderman 2007] Jesse Ruderman. *Introducing jsfunfuzz*, 2007. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz.

[Ruderman 2014] Jesse Ruderman. *Fuzzers love assertions*, 2014.

[Ruff 2013a] Nicolas Ruff. *Email*, 2013.

[Ruff 2013b] Nicolas "news0ft" Ruff. *Vers une certification ... utile*, 2013. http://news0ft.blogspot.fr/2013/07/vers-une-certification-utile.html.

[Rydstedt *et al.* 2010] Gustav Rydstedt, Elie Bursztein, Dan Boneh et Collin Jackson. *Busting frame busting: a study of clickjacking vulnerabilities at popular sites*. IEEE Oakland Web, vol. 2, 2010.

[Sankaranarayanan *et al.* 2013] Sriram Sankaranarayanan, Aleksandar Chakarov et Sumit Gulwani. *Static analysis for probabilistic programs: inferring whole program properties from finitely many paths*. In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, pages 447–458. ACM, 2013.

[Saxena *et al.* 2010a] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant et Dawn Song. *A symbolic execution framework for javascript*. In Security and Privacy (SP), 2010 IEEE Symposium on, pages 513–528. IEEE, 2010.

[Saxena *et al.* 2010b] Prateek Saxena, Steve Hanna, Pongsin Poosankam et Dawn Song. *FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications.* In NDSS, 2010.

[Scowen 1993] Roger S. Scowen. *Extended BNF — A generic base standard*. In SESP, 1993.

[Sekar 2009] R. Sekar. *An Efficient Blackbox Technique for Defeating Web Application Attacks*. In NDSS, 2009.

[Shahbaz & Groz 2009] Muzammil Shahbaz et Roland Groz. *Inferring Mealy Machines*. In FM: Formal Methods, pages 207–222. Springer, 2009.

[Shar & Tan 2012] Lwin Khin Shar et Hee Beng Kuan Tan. *Auditing the XSS defence features implemented in web application programs*. IET software, vol. 6, no. 4, pages 377–390, 2012.

[Shu & Lee 2007] Guoqiang Shu et David Lee. *Testing Security Properties of Protocol Implementations - a Machine Learning Based Approach*. In ICDCS. IEEE, 2007.

[Siemens 2014] Siemens. *e-health Solutions*, 2014. http://www.healthcare.siemens.com/hospital-it/e-health-solutions.

[Soltani 2013] Ashkan Soltani. *Here's everything you should know about NSA address book spying in one FAQ*, 2013. http://m.washingtonpost.com/blogs/the-switch/wp/2013/10/14/heres-everything-you-know-about-nsa-address-book-spying-in-one-faq/.

[Sotirov 2008] Alexander Sotirov. *Blackbox Reversing of XSS Filters*. In ReCon, 2008.

[SPaCIoS 2013] SPaCIoS. *Deliverable 5.3: Final Proof of Concept*, 2013.

[Stock *et al.* 2013] Ben Stock, Sebastian Lekies et Martin Johns. *25 Million Flows Later - Large-scale Detection of DOM-based XSS*. In 20th CCS. ACM, 2013.

[Stuttard 2007] Dafydd Stuttard. *Burp Suite*, 2007. http://portswigger.net/burp/.

[Su & Wassermann 2006] Zhendong Su et Gary Wassermann. *The essence of command injection attacks in web applications*. In POPL, 2006.

[Sun *et al.* 2009] Fangqi Sun, Liang Xu et Zhendong Su. *Client-Side Detection of XSS Worms by Monitoring Payload Propagation*. ESORICS, pages 539–554, 2009.

[Sutton & Linn 2004] Willie Sutton et Edward Linn. Where the money was: the memoirs of a bank robber. Broadway Books, New York, 2004.

[Takanen *et al.* 2008] Ari Takanen, Jared DeMott et Charles Miller. Fuzzing for software security testing and quality assurance. Artech House Publishers, 2008.

[Takanen 2012] Ari Takanen. *Fuzzing Embedded Devices*. In GreHack, 2012.

[Team 2012] PaX Team. *20 years of PaX*. In SSTIC, 2012. http://pax.grsecurity.net/docs/PaXTeam-SSTIC12-keynote-20-years-of-PaX.pdf.

[Thomas 2013] Steve Thomas. *DecryptoCat*, 2013. http://tobtu.com/decryptocat.php.

[Tonella *et al.* 2012] Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, Yue Jia, Kiran Lakhotia et Mark Harman. *Finding the Optimal Balance between Over and Under Approximation of Models Inferred from Execution Logs*. In ICST, 2012.

[Tripp *et al.* 2013] Omer Tripp, Omri Weisman et Lotem Guy. *Finding your way in the testing jungle: a learning approach to web security testing*. In ISSTA, pages 347–357. ACM, 2013.

[Trivedi *et al.* 2009] Kishor S Trivedi, Dong Seong Kim, Arpan Roy et Deep Medhi. *Dependability and security models*. In Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on, pages 11–20. IEEE, 2009.

[Utting & Legeard 2010] Mark Utting et Bruno Legeard. Practical model-based testing: a tools approach. Morgan Kaufmann, 2010.

[Valotta 2013] Rosario Valotta. *Fuzzing with DOM Level 2 and 3*. In DeepSec, 2013.

[Van Gundy & Chen 2009] Matthew Van Gundy et Hao Chen. *Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks.* In NDSS, 2009.

[Vogt *et al.* 2007] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kregel et Giovanni Vigna. *Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis*. In NDSS, 2007. http://publik.tuwien.ac.at/files/pub-inf_5310.pdf.

[Vuagnoux 2005] Martin Vuagnoux. *Autodafé: an act of software torture*. In 22nd CCC, 2005. http://autodafe.sourceforge.net/docs/autodafe.pdf.

[w3c 2009] w3c. *HTML5 IFrame Sandbox*, 2009. http://dev.w3.org/html5/markup/iframe.html.

[w3c 2012a] *Content Security Policy 1.0*, 2012. http://www.w3.org/TR/CSP/.

[W3C 2012b] W3C. *HTML5 Content Model*, 2012. http://www.w3.org/TR/html5/content-models.html.

[Wang *et al.* 2010] Yi-Hsun Wang, Ching-Hao Mao et Hahn-Ming Lee. *Structural Learning of Attack Vectors for Generating Mutated XSS Attacks*. Computing Research Repository, 2010.

[Wassermann *et al.* 2008] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura et Zhendong Su. *Dynamic test input generation for web applications*. In Proceedings of the 2008 international symposium on Software testing and analysis, pages 249–260. ACM, 2008.

[Wassermann 2008] Gary Michael Wassermann. *Techniques and Tools for Engineering Secure Web Applications*. PhD thesis, UC Davis, 2008.

[Weinberger *et al.* 2011a] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin et D. Song. *A systematic analysis of XSS sanitization in web application frameworks*. In ESORICS, pages 150–171. Springer, 2011.

[Weinberger *et al.* 2011b] Joel Weinberger, Adam Barth et Dawn Song. *Towards client-side HTML security policies*. In HOTSEC. USENIX, 2011.

[Wen *et al.* 2013] Guanxing Wen, Yuqing Zhang, Qixu Liu et Dingning Yang. *Fuzzing the ActionScript Virtual Machine*. In CCS. ACM, 2013.

[Wikipedia 2006] Wikipedia. *Cryptographic nonce*, 2006. http://en.wikipedia.org/wiki/Cryptographic_nonce.

[Woo *et al.* 2013] Maverick Woo, Sang Kil Cha, Samantha Gottlieb et David Brumley. *Scheduling black-box mutational fuzzing*. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 511–522. ACM, 2013.

[Xu *et al.* 2006] Wei Xu, Sandeep Bhatkar et R Sekar. *Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks*. In Proceedings of the 15th USENIX Security Symposium, pages 121–136, 2006.

[Yarom 1999] Yuval Yarom. *Method of relocating the stack in a computer system for preventing overrate by an exploit program*. US Patent 5,949,973, 1999. http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=5949973&KC=&FT=E&locale=en_EP.

[Yoda 2001] Yoda. *Shadderpoint*, 2001.

[Zalewski & Heinen 2009] Michal Zalewski et Niels Heinen. *SkipFish - web vulnerability scanner*, 2009. http://code.google.com/p/skipfish/.

[Zalewski 2011a] Michal Zalewski. *Announcing CrossFuzz*, 2011. http://lcamtuf.blogspot.fr/2011/01/announcing-crossfuzz-potential-0-day-in.html.

[Zalewski 2011b] Michal Zalewski. *Postcards from the post-XSS world*, 2011. http://lcamtuf.coredump.cx/postxss/.

[Zalewski 2011c] Michal “lcamtuf” Zalewski. *An origin is forever*, 2011. http://lcamtuf.blogspot.fr/2011/10/origin-is-forever.html.

[ZentrixPlus 2013] ZentrixPlus. *eBay Sec. Hall of Fame*, 2013. http://zentrixplus.net/blog/ebay-security-researchers-hall-of-fame-hof/.

List of Figures

List of Tables

List of Algorithms

Listings

List of Acronyms

AIG Attack Input Grammar 40
Ajax Asynchronous Javascript And XML 107
BFS Breadth First Search 62
BOF Buffer Overflow 106
CFM Control Flow Model 37
CSP Content Security Policy 114
CSRF Cross-Site Request Forgery 13
CSS Cascading Style Sheets 154
CTFM Control and Taint Flow Model
DFS Depth First Search 47
DNS Domain Name System 68
DOM Document Object Model 24
DOM-XSS DOM Cross Site Scripting (a.k.a. Type-0 XSS) 106
FP False Positive 98
FQDN Fully Qualified Domain Name 68
FSM Finite State Machine 107
GA Genetic Algorithm 74
HI-CFG Hybrid Control Flow Graph 70
HTML HyperText Markup Language 40
HTTP Hypertext Transfer Protocol 23
JS JavaScript 22
KF KameleonFuzz 99
MCV Memory Corruption Vulnerability 106
m-XSS Mutation based Cross Site Scripting 88
NDV Non-Deterministic Values 44
TAP Taint Aware Pattern
TAT Taint-Aware Tree 41
TCP Transmission Control Protocol 68
TP True Positive 98
URL Uniform Resource Locator 23
WA Web Application 28
WAF Web Application Firewall 34

APPENDIX A

Web Scanners Configuration

We here list the main settings used during experiments .

- **Wapiti 2.20**: `-m "-all,xss"`
- **w3af 1.2 kali 1.0**:

```
misc-settings
set maxThreads 1
set maxDepth 200
set maxDiscoveryTime 18000
back
plugins
discovery webSpider
discovery config webSpider
    set onlyForward True
    back
audit xss
audit config xss
    set numberOfChecks 3
    back
back
start
```

Listing A.1: w3af configuration

- **SkipFish 2.10b**: `-Y -Z -m 10 -k 18000`
- **LigRE**: The model annotation limits are as follows:
  - minimal reflection length: 6 characters
  - maximal input sequences length: 8 HTTP requests

  The Fuzzing parameters are:
  - fuzzer : w3af
  - pass_the_context : cookie
- common parameters in **KameleonFuzz 2013-08-31** are mentioned in Table A.1

Parameter	Default Value
`LigRE.targeted_reflections` – The percentage of reflections that KameleonFuzz will focus on. LigREorders them in descending order of potential interest [Duchène *et al.* 2013d].	0.8
`GA.population_size` – The size of the population i.e., the number of individuals. The actual amount is this value times the number of targeted LigREreflections.	5
`GA.elitism` – Number of individuals having the highest fitness score that are kept for the next generation.	4
`GA.mutation_proba` – The probability to apply a mutation operator on a new child.	0.5
`GA.crossover_num_exchanges` – Number of exchanges performed by the crossover operator. One exchange means a two points crossover i.e., for the whole sub-tree of the exchanged grammar (non)-terminal.	1

Table A.1: Common parameters in KameleonFuzz and their default values. See Section 6.4.4 on how we chose those default values.

```
1  <!DOCTYPE RootElement SYSTEM "RootElement.dtd">
2  <KameleonFuzzConfig>
3    <FuzzingPruning>
4      <param name="max_index_of__do_not_follow" value="200" />
5      <param name="do_not_fuzz[0]" value="PARAM_NAME pw" />
6      <param name="do_not_fuzz[1]" value="PARAM_NAME uid" />
7    </FuzzingPruning>
8
9    <Random>
10     <param name="seed.inference" value="1361641868" />
11       <!--
12       Seed to feed the Prime Random Number Generator (useful for
             tests replication). System time is used if no seed is
             provided
13       -->
14     <param name="seed.annotation" value="" />
15     <param name="seed.fuzzing" value="" />
16   </Random>
17
18   <Logging>
19     <!--param name="loglevel" value="DEBUG" /-->
20     <param name="loglevel" value="INFO" />
21     <!-- DEBUG, ERROR, WARN, NOTICE, INFO, NONE -->
22   </Logging>
23
```

```
24    <EvolutionaryAlgorithmConfig>
25      <param name="numberOfPoolsOrIslands" value="1" />
26      <param name="
            percentageOfPopulationMigratingFromOneIslandOrPoolToAnotherOne
            " value="0.04" />
27
28      <param name="PopulationSize" value="10" />
29      <param name="PopulationTakingPartInRecombination" value="1.00"
             />
30        <!-- FLOOR of this value * population_size will be choosen:
             how many % of the BEST individuals do we consider for
             crossover -->
31      <param name="Elitism" value="2" />
32      <param name="crossoverNodeSelectionAndNumberOfPointsStrategy"
            value="samePrefix_and_1NodeRandom" />
33        <!-- strategies: samePrefix_and_1NodeRandom,
             samePrefix_and_1NodeFirstFromRoot -->
34      <param name="mutationRate" value="0.5" />
35      <param name="mutationStrategy" value="random" />
36        <!-- other values of mutationStrategy include # most
             frequent value first ... within nodes "close" from leafs
              -->
37      <param name="firstGenerationInputParamSelectOneInXPercents"
            value="0.8" />
38      <param name="modelMaxDepthForPrefixingInputSequences" value="7
            " />
39    </EvolutionaryAlgorithmConfig>
40
41    <Fitness> <!-- the higher the more important the weight will be
        -->
42      <param name="number_of_classes_injected_vs_sent" value="2" />
43      <param name="string_distance" value="2" />
44      <param name="number_of_tainted_nodes" value="3" />
45      <param name="
            number_of_nodes_between_fuzzed_input_sending_and_reflection
            " value="2" />
46
47      <param name="number_of_unique_states_from_start_node" value="
            0.5" />
48      <param name="how_well_formed_wrt_HTML_is_the_output" value="
            0.5" />
49
50      <param name="new_output_symbol_discovered" value="5" />
51      <param name="percentage_of_expected_output_symbols" value="3"
            />
52      <param name="
            number_of_different_macro_states_between_fuzzed_value_submission_and_reflection
            " value="1" />
53        <param name="singularity_on_fuzzed_input_param_value" value=
              "4" />
54        <param name="singularity_on_input_sequence" value="3" />
55    </Fitness>
56
57    <InternalTests>
```

```
58      <param name="run" value="True" />
59    </InternalTests>
60
61
62    <Crawling>
63      <param name="stop.when_conjecture_is_complete" value="True" />
64      <param name="stop.max_duration" value="0" /> <!-- in seconds
            -->
65      <param name="stop.max_num_of_http_requests" value="200" />
66      <param name="longest_path.max_times_per_edge" value="7" />
67      <param name="longest_path.max_times_per_node" value="7" />
68      <param name="longest_path.max_depth_of_subsequence" value="10"
            />
69      <param name="stop.
            minimum_number_of_times_we_went_trough_each_transition"
            value="7" />
70      <param name="stop.minimum_number_for_all_clusters" value="7" /
            >
71      <param name="stop.
            number_of_times_to_go_through_each_link_and_no_new_page_discovered
            " value="7" />
72      <param name="
            max_explored_times_for_a_transition_for_djisktra_computation
            " value="7" />
73      <param name="max_depth_of_input_sequence" value="170" />
74      <param name="conjecture.
            max_consecutive_potential_contradictions" value="4" />
75      <param name="follow_external_links" value="none" />
76      <param name="conjecture.strategy" value="from_root" />
77      <param name="
            once_emergency_sequence_executed_forbid_emergency_for_x_requests
            " value="2" />
78      <param name="save_model_in_config_folder" value="True" />
79      <param name="load_last_model_in_config_folder" value="True" />
80      <param name="folder_where_to_save_progression" value="logs/
            gruyere/0_inferred_models/" />
81
82      <param name="folder_where_to_save_and_load" value="logs/" />
83        <!-- which folder will be used to check for a previously
              annotated model -->
84        <!-- or where to save it -->
85      <param name="crawled_model_filename_suffix" value="
            crawled_model.json" />
86
87      <!-- we should here indicate all the nonce fields that could
            lead to identifying different states -->
88      <param name="prefix_tree.fields_to_ignore" value="form.hidden.
            nonce" />
89        <!-- if it is a form request, then we will ignore any field
              that is of type hidden and of name nonce -->
90      <param name="http_request_comparison.fields_to_ignore" value="
            params_structured.nonce" />
91        <!-- eg: for each GET or POST parameter, we will ignore any
              field that names is nonce -->
```

```
92
93      <param name="performance.memory.
            execute_only_x_last_navigation_sequences" value="20" />
94      <param name="performance.dot.
            discard_task_if_not_started_within_x_seconds" value="100"
            />
95      <param name="performance.conjecture.
            skip_executing_whole_sequences_on_infered_FSM" value="True
            " />
96
97      <param name="dot.skip_saving_files_and_producing_svg" value="
            False" />
98        <!-- can be useful for speeding execution -->
99
100     <param name="dot.infered_FSM_filename" value="infered_fsm.svg"
             />
101     <param name="dot.navigation_graph_filename" value="
            navigation_graph.svg" />
102     <param name="dot.skip_outputting_non_yet_explored_spiderlinks"
             value="True" />
103     <param name="dot.max_output_times" value="2" /> <!-- to avoid
            repetition -->
104     <param name="dot.max_simulatneous_processes" value="2" />
105       <!-- to make use of multi-core computers-->
106       <!-- be aware that httpd and chrome will already use 1 each
              one-->
107
108     <param name="dot.enable_html_tags" value="False" />
109     <param name="conjecture.
            assumes_non_determinism_is_due_to_wrong_macro_state_hypothesis
            " value="True" />
110     <param name="conjecture.wrong_macro_state.
            assumes_it_is_the_latest_change_that_went_to_an_already_known_macro_state_whereas_it_i
            " value="True" />
111     <param name="conjecture.die_when_wrong_prefix_tree" value="
            False" />
112       <!-- set to false for gruyere -->
113     <param name="conjecture.wrong_prefix_tree.assumes_state_change
            " value="True" />
114       <!-- set to True for gruyere -->
115
116     <!-- weights for confidence function -->
117     <param name="confidence.weight.
            number_of_unexplored_links_on_model_for_current_state_so_far
            " value="2" />
118     <param name="confidence.weight.nodes_from_root__shortest_path"
             value="1" />
119     <param name="confidence.weight.nodes_from_root__actual_path"
            value="0" />
120     <param name="confidence.weight.
            number_of_explored_links_that_have_same_spiderlinks_as_one_who_helped_determining_a_st
            " value="0" />
121     <param name="confidence.weight.
            number_of_unexplored_links_that_have_same_spiderlinks_as_one_who_helped_determining_a_
```

```
        " value="0" />
122     <param name="confidence.weight.number_of_incoming_edges" value
            ="0.1" />
123
124     <param name="conjecture.
            consider_only_first_spiderpage_for_confidence_computation"
             value="True" />
125
126
127     <param name="raiseExceptionWhenEmptyForms" value="False" />
128       <!-- if False, at least we will print a WARNING -->
129     <param name="infered_model.filename_suffix" value="
            infered_model.pickle" />
130     <param name="infered_model.
            number_of_folders_in_which_to_search_for" value="3000" />
131       <!-- we search in the last 5 recent log folders -->
132
133     <param name="skip_outputting_graphs" value="False" />
134
135     <param name="die_in_case_of_exception" value="False" />
136     <param name="exit_after_inference" value="True" />
137     <param name="stopping.finish_outputting_graphviz_files" value=
            "True" />
138     <param name="stopping.timeout" value="200" />
139
140     <param name="state_change.score.num_of_parameters" value="0.3"
             />
141     <param name="state_change.score.boost_if_POST_request" value="
            0.18" />
142     <param name="state_change.score.boost_if_GET_request" value="
            0.10" />
143     <param name="state_change.score.distance_num_of_requests"
            value="0.3" />
144       <!-- default value is 1 -->
145     <param name="state_change.score.num_of_contradictions" value="
            2" />
146       <!-- default value is 1 -->
147
148     <param name="cluster.weights.min_number_of_leaves" value="5" /
            >
149       <!-- value used for p0wn_me,wordepress,webgoat  is 5 -->
150     <param name="navigation.score.weight.
            likelyhood_to_detect_a_state_change" value="0.3" />
151     <param name="navigation.score.weight.
            if_new_page_boost_already_seen_requests" value="0.6" />
152     <param name="navigation.score.weight.never_seen_boost" value="
            0.4" />
153     <param name="navigation.score.weight.
            num_of_times_recently_performed" value="1.0" />
154     <param name="navigation.score.
            num_of_times_recently_performed__to_take_into_account"
            value="1" />
155     <param name="navigation.score.boost_if_POST_request" value="
            0.16" />
```

```
156       <param name="navigation.score.boost_if_GET_request" value="
              0.10" />
157       <param name="navigation.score.weight.
              number_of_artificial_values" value="0.1" />
158       <param name="navigation.score.weight.tester_provided_values"
              value="0.3" />
159       <param name="conjecture.
              consider_only_first_spiderpage_for_confidence_computation"
               value="True" />
160
161       <param name="navigating.
              go_back_to_root_when_currently_accessible_graph_is_complete_
              " value="True" />
162       <param name="pickle.sys.recursionlimit.increase_factor" value=
              "10" />
163     </Crawling>
164
165     <CrawlingFieldsValues>
166       <param name="subset_matching_enabled" value="true" />
167       <param name="
              number_of_expansion_to_consider_before_smartfilling" value
              ="15" />
168       <param name="expansion_sorting_order" value="
              no_more_polar_bears" />
169         <!-- reversed or anything else-->
170       <param name="
              expand_prefilled_values_with_one_of_length_greater_of_equal_than_min_length_to_infer_t
              " value="True" />
171     </CrawlingFieldsValues>
172
173
174     <CrawlingSlicing>
175       <!-- prevent some requests to be performed -->
176       <!-- works in a black-listing mode -->
177       <!-- useful for pruning the inference and the fuzzing -->
178
179       <param name="max_index_of__do_not_follow" value="10" />
180       <param name="do_not_follow[0]" value="GET /xss-type-1/?action=
              logout" encoding="plain" />
181       <param name="do_not_follow[3]" value="FORM_FIELD_VALUE
              orlandopassword" encoding="plain" />
182       <param name="do_not_follow[4]" value="GET /xss-type-1/?action=
              auth" encoding="plain" />
183       <param name="do_not_follow[5]" value="FORM_FIELD_VALUE Create
              account" />
184       <param name="do_not_follow[6]" value="FORM_FIELD_VALUE Upload"
               />
185     </CrawlingSlicing>
186
187
188     <HFuzz>
189       <param name="Fuzzer" value="w3af" />
190       <param name="credentials" value="cookie" />
191     </HFuzz>
```

```
192
193
194   <ModelAnnotation>
195     <param name="ignoreOutputSymbols" value="True" />
196     <param name="modelExplorationMethod" value="
            breath_first_prefix_1by1_bounded" />
197       <!-- values include:
198         - breath_first_prefix_1by1_bounded
199         - random_bounded:
200       NOTE: for both, the length of input sequences is bounded by
              modelMaxInputSequencesLength
201       -->
202     <param name="random_bounded.
            max_attempts_to_generate_input_sequences" value="6" />
203       <!--
204       60
205       -->
206     <param name="max_number_of_times_per_each_state" value="1" />
207       <!-- limits loop in the model. values:
208         * (not implemented yet)
209         or 1,2,3...
210       -->
211     <param name="method" value="efficient_substring" />
212       <!-- possible values:
213         - efficient_substring
214         - edit_distance
215       -->
216     <param name="efficient_substring.min_string_length" value="6"
            />
217       <!-- only input parameters with value of at least xx
              characters will be considered for approximate taint
              computation -->
218
219     <param name="modelMaxBackwardDepthForAnnotatingTaint" value="7
            " />
220       <!-- should be greater or equal than the value of
              modelMaxDepthForPrefixingInputSequences -->
221     <param name="modelMaxInputSequencesLength" value="8" />
222       <!-- should be greater or equal than the value of
              modelMaxBackwardDepthForAnnotatingTaint -->
223
224     <!-- param name="maxNumberOfGeneratedInputSequences" value="60
            " /-->
225
226     <param name="save_annotated_model_in_config_folder" value="
            True" />
227       <!-- values: True or anything else -->
228     <param name="load_last_annotated_model_in_config_folder" value
            ="True" />
229
230     <param name="folder_where_to_save_and_load" value="logs/
            gruyere/1_annotated_models" />
231       <!-- which folder will be used to check for a previously
              annotated model -->
```

```
232       <!-- or where to save it -->
233
234     <param name="annotated_model_filename_suffix" value="
            annotated_model.pickle" />
235
236     <param name="graphviz.generate" value="True" />
237     <param name="graphviz.in_folder" value="logs/gruyere/1
            _annotated_models" />
238     <param name="graphviz.representation_type" value="
            concise_from_and_to_transition" />
239       <!-- values:
240         steps_123
241         concise_from_and_to_transition
242       -->
243
244     <param name="exit_after_model_annotation" value="False" />
245     <param name="list_reflections" value="True" />
246     <param name="list_reflections.exit_after" value="False" />
247   </ModelAnnotation>
248
249   <TaintInferenceConfig>
250     <param name="useExactStringMatchingFirst" value="true" /> <!--
            true, false -->
251     <param name="distance" value="sekar_basic" />
252       <!--
253         edit_distance, as described in Sekar's paper
254         sekar_optimized
255       -->
256
257     <param name="costInsertion" value="1" />
258     <param name="costDeletion" value="1.9" />
259     <param name="minimumLengthOfTaintedStrings" value="6" />
260     <param name="sekar.minLengthOfTaintedSubstringsInOutput" value
            ="6" />
261
262     <param name="propagation.minLengthOfOutputSubstring" value="6"
            />
263     <param name="propagation.minLengthOfInputSubstring" value="6"
            />
264
265     <param name="maxDistanceToTaint" value="1" /> <!-- NOT USED --
            >
266
267     <!-- those values work well for substring taint propagation
268       DOMTaintIfNodeAlone = 1-0.15
269       DOMTaintIfNodeWithBrother = 1-0.06
270       DOMTaintMaxNodeDistance = 2
271     -->
272     <param name="DOMTaintIfNodeAlone" value="0.65" />
273     <param name="DOMTaintIfNodeWithBrother" value="0.40" />
274     <param name="DOMTaintMaxNodeDistance" value="2" />
275     <param name="TaintPropagationOnDOM"  value="substring" />
276       <!-- substring or sekar_basic -->
```

```
277       <!-- substring is faster, but more false negative on taint
              propagation, thus on fault verdict also -->
278
279     <param name="assertSamePageIsReached" value="False" />
280   </TaintInferenceConfig>
281
282   <Fuzzing>
283     <param name="input_sequences.generating_strategy" value="
          ShortestPath" />
284     <param name="
          max_number_of_fuzzed_input_parameters_per_individual"
          value="1" />
285     <param name="
          max_number_of_fuzzed_input_parameters_per_transition"
          value="1" />
286     <param name="
          during_fuzzing_do_not_require_reaching_same_state_as_in_model
          " value="True" />
287
288   </Fuzzing>
289
290   <VulnerabilityToSearch>
291     <param name="type" value="HTTP-HTML-XSS-reflected" />
292     <param name="AttackInputGrammar" value="./KameleonFuzz/Grammar
          /HTTP/XSS/attack_grammar.kfgrammar" />
293   </VulnerabilityToSearch>
294
295
296   <!-- if any of those conditions is evaluated to true, then the
        program stops -->
297   <StoppingConditions>
298     <param name="numberOfFoundFaults" value="1" />
299     <param name="max_duration" value="60" /> <!-- in seconds -->
300     <param name="EA_MaxPopulationGenerations" value="200" />
301   </StoppingConditions>
302
303
304   <SUTConfig>
305
306     <param name="interface.protocol" value="HTTP" />
307     <param name="interface.host" value="127.0.0.1" />
308     <param name="interface.port" value="8008" />
309     <param name="interface.baseHREF" value="/2497762752997571069/"
           />
310     <param name="startPage" value="" />
311     <param name="Reset.HTTP_Request.action" value="GET
          /2497762752997571069/reset" />
312     <param name="Reset.HTTP_Request.verification" value="(.*)
          Server reset to default values...(.*)" />
313
314     <if condition="os.name==’nt’">
315       <param name="reset.command.action" value="C:/Users/php.exe -
            c C:/Users/fabite/Documents/EasyPHP-5.3.8.1/apache C:/
            Users/fabite/Dropbox/git/KameleonFuzz/KameleonFuzz/
```

```
            config/gruyere/script_reset.php &quote;coucou&quote;" />
316       </if>
317
318     </SUTConfig>
319
320
321   </KameleonFuzzConfig>
```

Listing A.2: Extract of `config.xml` configuration file

APPENDIX B

KameleonFuzz: List of Taint Aware tree Patterns

We incorporate in KameleonFuzz the list of default TAP illustrated in Table B.1

Name	Reflection Context (see Table 4.5)	Grammar	TAP
tainted event handler and action	inside an attribute value	HTML	.+ | attributes | (onerror‖onload‖...) | .*.+.*
tainted script tag and content	• outside a tag • inside a textarea	HTML	script | children | .+
tainted url / src with a script pseudo-protocol	inside a src/href attribute value	HTML	.+ | attributes | (src‖href) | (javascript:|vbscript:).*.+.*

Table B.1: KF: List of Implemented Taint Aware tree Patterns (TAP)

We are aware that this list does not cover all cases of XSS. In Table B.2, we list additional TAP that would be of interest.

Name	**Reflection Context**	**Grammar**	**TAP**
Cascading Style Sheets (CSS) escape	inside a CSS value	CSS	`.+` / attributes / `(onerror||onload||...)` / `.*.+.*`
CSS URL handler	inside a CSS url value	CSS	`.+` / attributes / `(src||href)` / `(javascript:|vbscript:).*.+.*`
JS escape	inside a JS value	JS	`.+` / expression / `(.*.+)` AND `.+` / instructions / `(.+.*)`

Table B.2: KF: List of Additional TAP to be Implemented

APPENDIX C

0-day Found XSS Vulnerabilities

Two of the 0-days XSS discovered by KameleonFuzz

We illustrate two 0-day XSS that KF uncovered:

- Elgg: Type-2 XSS (CVE-2013-7297). The control flow inference, the reflections, and the taint aware patterns permit detecting this XSS present since several versions.

- SFR BOX NB6-MAIN-R3.3.4: 39 Type-1 XSS (CVE-2014-1599). The control flow inference permits navigating sufficiently deep enough in the application.

Elgg 1.8.13: 1 Type-2 XSS (CVE-2013-7297)

Elgg is an open-source social network notably used by the University of Florida and the Australian governement. Elgg 1.8.13 suffers from an unfiltered Type-2 XSS in the website field of the user (see Figure C.1). This vulnerability has been responsibly reported [Duchène 2013a].

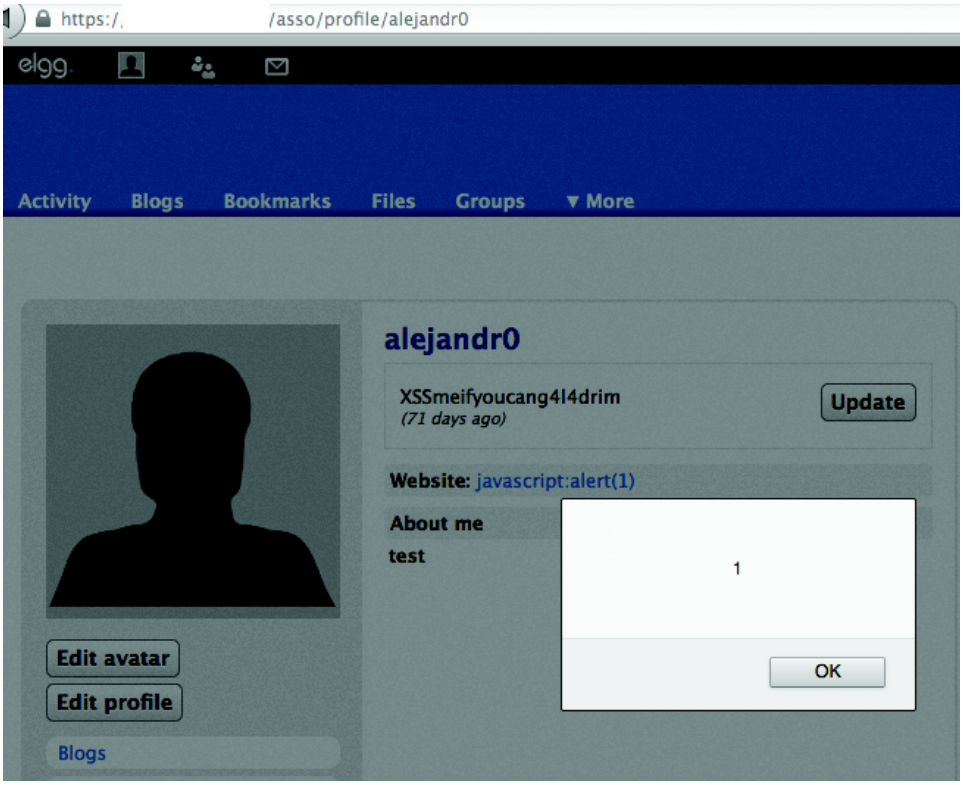


Figure C.1: Elgg 1.8.13 – Type-2 XSS in the website field (CVE-2013-7297)

SFR BOX NB6-MAIN-R3.3.4: 39 Type-1 XSS (CVE-2014-1599)

SFR is the french Vodafone (estimated DSL user base of 5.2 Millions). The affected product is SFR BOX NB6-MAIN-R3.3.4. It suffers from 39 unfiltered Type-1 XSS. Some are illustrated in Figure C.2. These vulnerabilities have been reported using responsible disclosure process [Duchène 2014a].

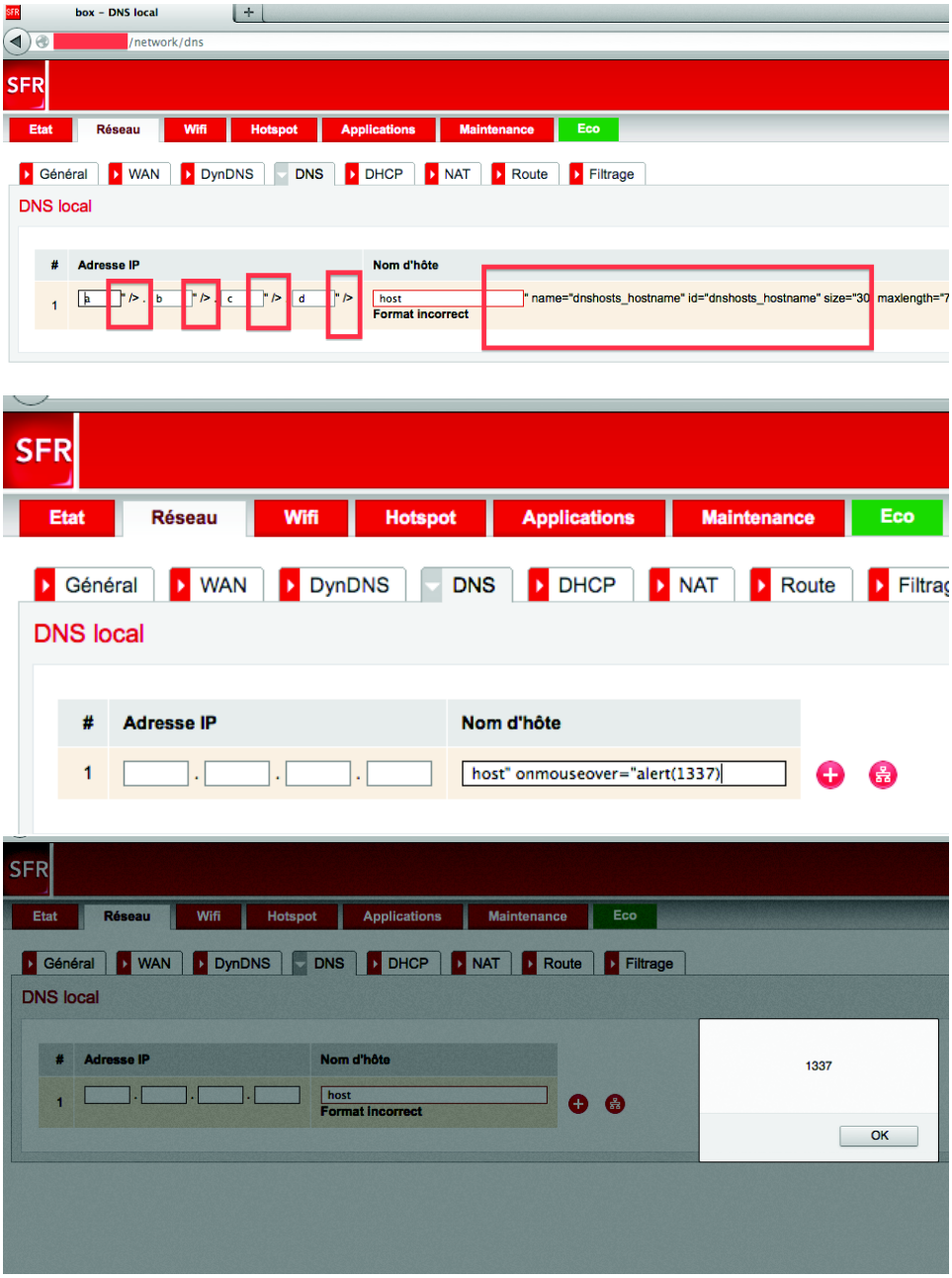


Figure C.2: Examples of Type-1 XSS in SFR BOX NB6-MAIN-R3.3.4

Vulnerabilities

- /network/dns: 5 non-filtered Type-1 XSS
- /network/dhcp: 6 non-filtered Type-1 XSS
- /network/nat: 7 non-filtered Type-1 XSS

- /network/route: 12 non-filtered Type-1 XSS
- /wifi/config: 1 non-filtered Type-1 XSS
- /network/lan: 8 non-filtered Type-1 XSS

Exploitation hypotheses The requirements for such exploits to work are:

- the user is already logged-in (or tricked by SE techniques to authenticate in the interface of his web router)
- ip address of the SFR Box router is known (most users use the default settings: 192.168.1.1/24)

Example of exploitation scenario If a user is tricked into authenticating into its interface, an attacker can XSS the user, and thus getting read and write access to the router configuration webpages. Such as scenario is mainly possible due to non filtered reflections (mainly Type-1 / reflected) and the lack of Content Security Policy. Moreover, no anti-CSRF token such as view-states are present, thus there is the possibility of modifying the routing tables even without an XSS, if the user is authenticated in the box.

A non limitative list of actions include:

- getting authentication credentials (wireless, DSL credentials)
- rebooting the router
- modifying the route table (thus possibility of content injection if an attacker controlled server is on the route)
- DDOSing a target with numerous XSS'ed clients

Examples of Vulnerabilities that KameleonFuzz is unable to detect

We illustrate three vulnerabilities that KF is unable to detect as of today:

- `mega.co.nz`: cross-domain double parameter HTML injection. KameleonFuzz only supports one fuzzed input parameter value at a time.
- Evernote: unconstrained URL handler (CVE-2014-1404). KameleonFuzz test drivers only support websites ; a CTFM containing reflections into libc parameters is required to detect this vulnerability.
- Siemens-Home: Two type-2 Reflections in HTML context result in IE-7 XSS. KameleonFuzz only supports one fuzzed input parameter value at a time.

`http://mega.co.nz`: Phishing with Cross-Domain HTML Content Injection in emails

Two parameters entered during the subscription transition (see Figure C.3) are reflected in the HTML email sent by the mega servers (see the HTML code in Figure C.4 and the rendering in Figure C.5).

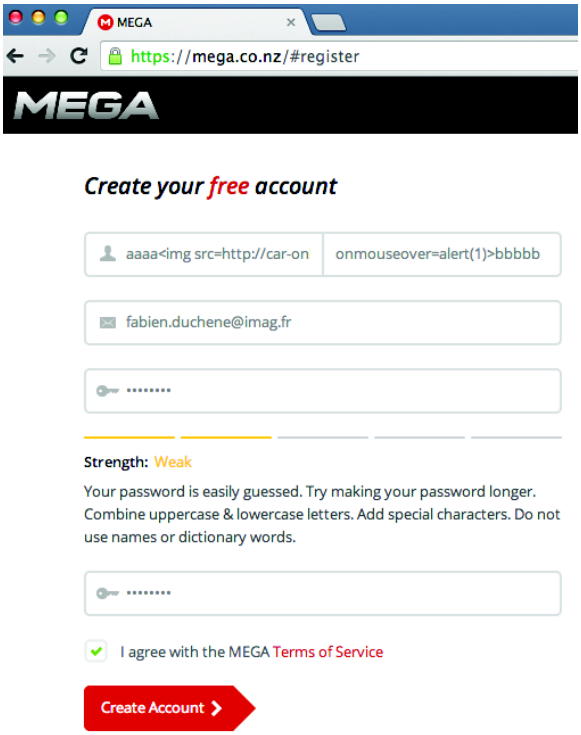


Figure C.3: mega: source transition

Evernote 5.4.4 (402282): unconstrained file handler (CVE-2014-1404)

I reported using a responsible disclosure process an Unconstrained file handler which permits attacker controlled code execution [Duchène 2014e].

We can provide a local executable file in the URL field (¡source-url¿ tag) of a note. Once the note is opened or imported:

- on Mac OS X, if a user is tricked into clicking on info, open, it results in attacker controlled shell command execution. No warning whatsoever is displayed to the user
- on Windows, a warning to the user is presented, but this still is a risk on this platform and should be prevented

```
To: fabien.duchene@imag.fr
From: MEGA <support@mega.co.nz>
Subject: MEGA Signup
Content-Type: multipart/related; boundary=xead6ad29639913b1
Message-Id: <20140129185129.3C2571680A28@mail5.mega.co.nz>
Date: Wed, 29 Jan 2014 18:51:29 +0000 (UTC)
X-Greylist: Sender IP whitelisted, not delayed by milter-greylist-4.2.2 (romir
X-Greylist: Delayed for 00:17:04 by milter-greylist-4.2.2 (rominette.imag.fr [

--xead6ad29639913b1
Content-Type: multipart/alternative; boundary=y4c94c4a75908ce0f

--y4c94c4a75908ce0f
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 7bit

Dear aaaa<img src=http://car-online.fr/images/mega-co-nz_phish.jpg
onmouseover=alert(1)>bbbbb,

Welcome to MEGA! Please click on the following link to confirm your free MEGA
account:

https://mega.co.nz/#confirmHGUv6PDvA4Ub_7saAJlIaLahYvNjXgmzM_XZgQ6kwGRqbzoc4AA
i_fYFyvXbGZQ

Best regards,

Team MEGA

--y4c94c4a75908ce0f
Content-Type: text/html; charset=UTF-8
Content-Transfer-Encoding: quoted-printable

<p style=3D" margin:0; padding:20px 30px 0 30px; font-family: 'Open Sans', =
Arial; font-size:13px; color:#525252;">Dear aaaa<img src=3Dhttp://car-onlin=
e.fr/images/mega-co-nz_phish.jpg  onmouseover=3Dalert(1)>bbbbb,</p><p style=
=3D" margin:0; padding:20px 30px 0 30px; font-family: 'Open Sans', Arial; f=
ont-size:13px; color:#525252;">Welcome to MEGA! Please click on the followi=
```

Figure C.4: mega: reflection transition, HTML source code

The creation of such files can be automated via the creation of XML document conform to the Evernote DTD: `http://xml.evernote.com/pub/evernote-export3.dtd` and via a specially crafted source-url tag: `<source-url>file:////bin/sh</source-url>`

I tested this vulnerability on the following versions: Evernote 5.4.4 and Evernote 5.5. 402941 direct, Mac OS X (see Figure C.6) and on Windows (see Figure C.7).

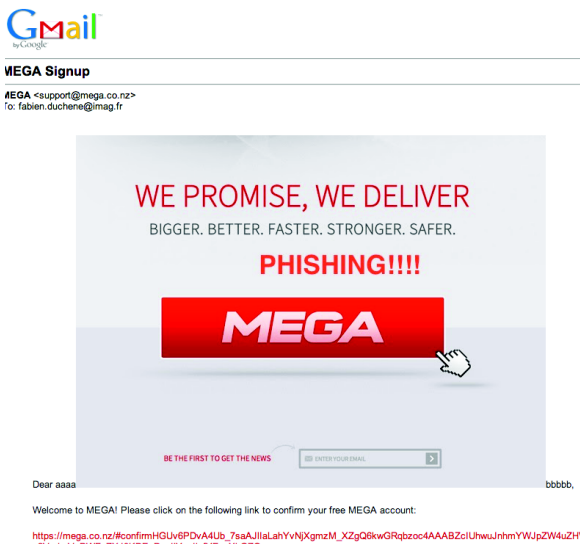


Figure C.5: mega: reflection transition, rendered webpage

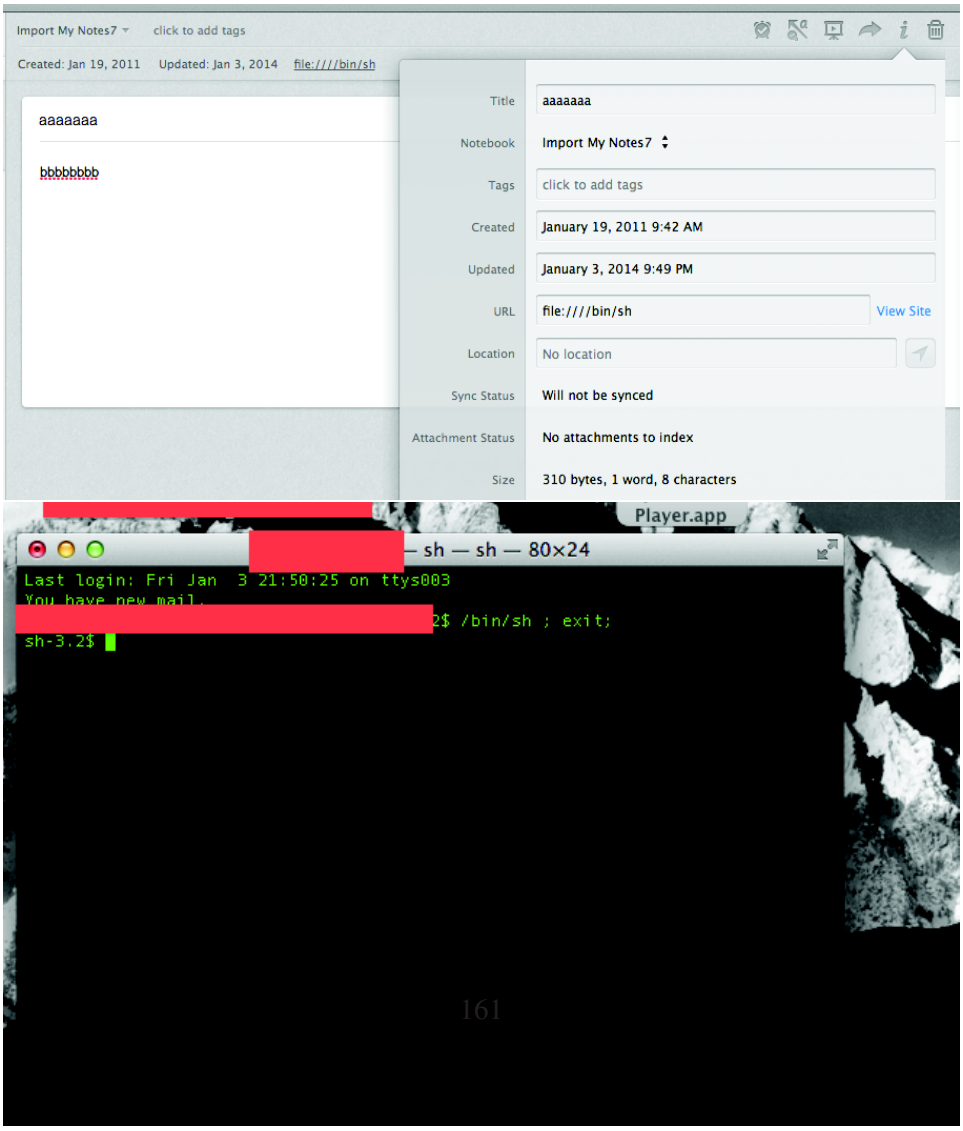


Figure C.6: Evernote CVE-2014-1404: on Mac OS X

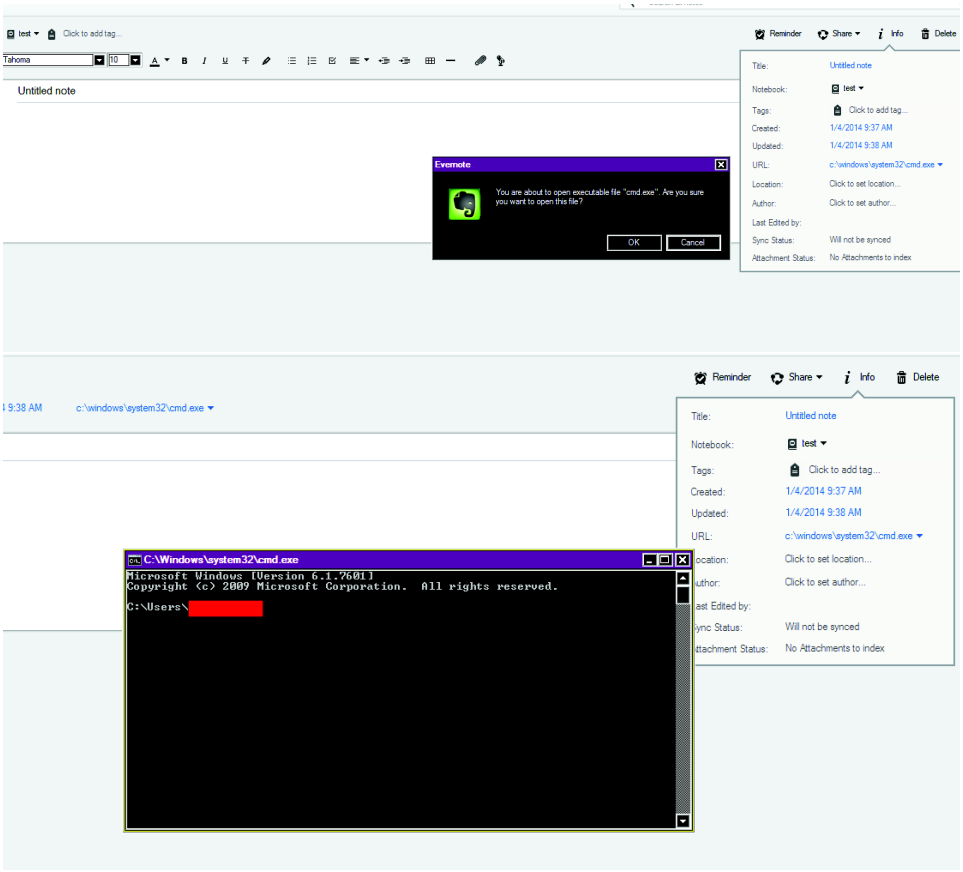


Figure C.7: Evernote CVE-2014-1404 on Windows: a warning is displayed though

Siemens-Home: 1 Type-1 XSS, 2 Type-2 XSS including 1 IE7-IE10 specific XSS [Duchene 2014b]

Reflection in a JS context An unauthenticated webpage had a Type-1 reflection, as illustrated in Figure C.8 and Figure C.9. The server-side sanitizers were targeting reflections in an HTML context.

Figure C.8: Siemens–Type-1 Reflection in a JS context - code

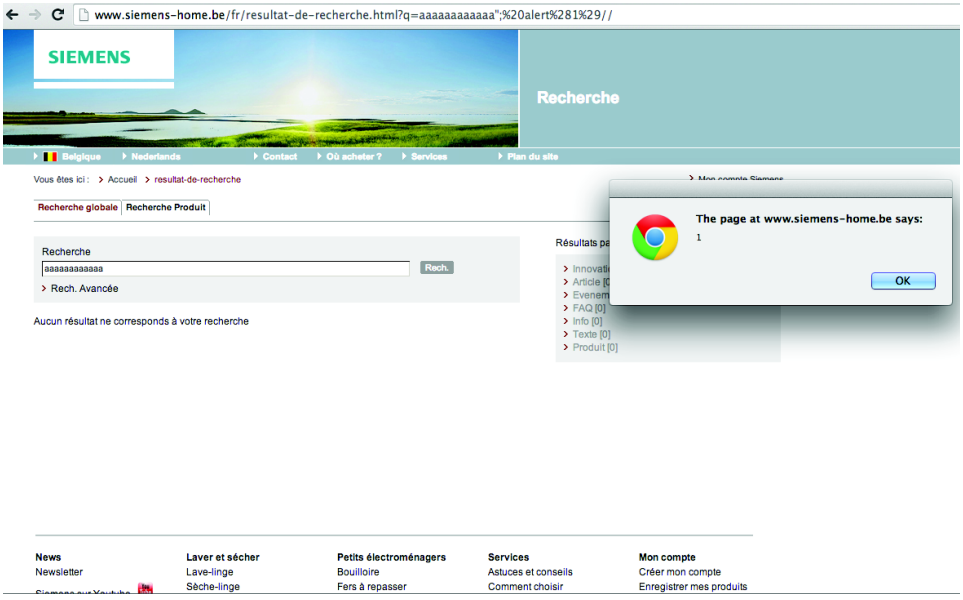


Figure C.9: Siemens–Type-1 Reflection in a JS context - execution

Type-2 Reflection in a JS context A Type-2 reflection exists from the `city` field (Figure C.10) into the `s.state` JS sink (Figure C.11) results in an exploitable XSS (Figure C.12).

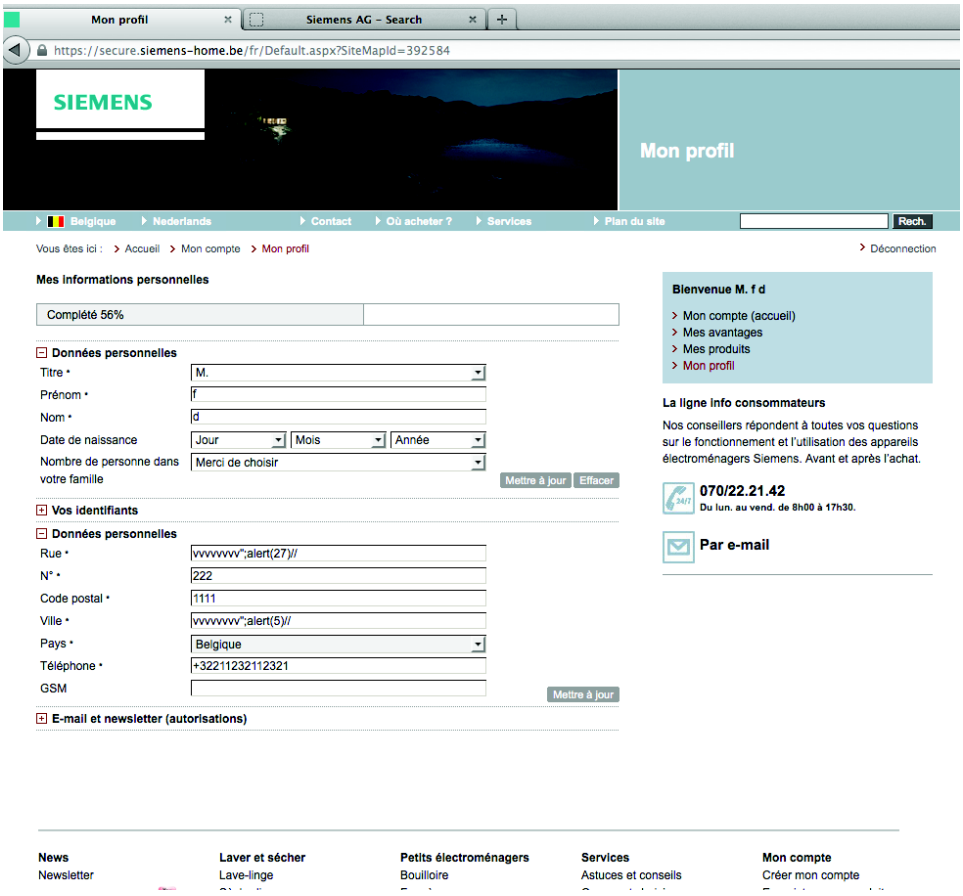


Figure C.10: Siemens–Type-2 Reflection in a JS context - source

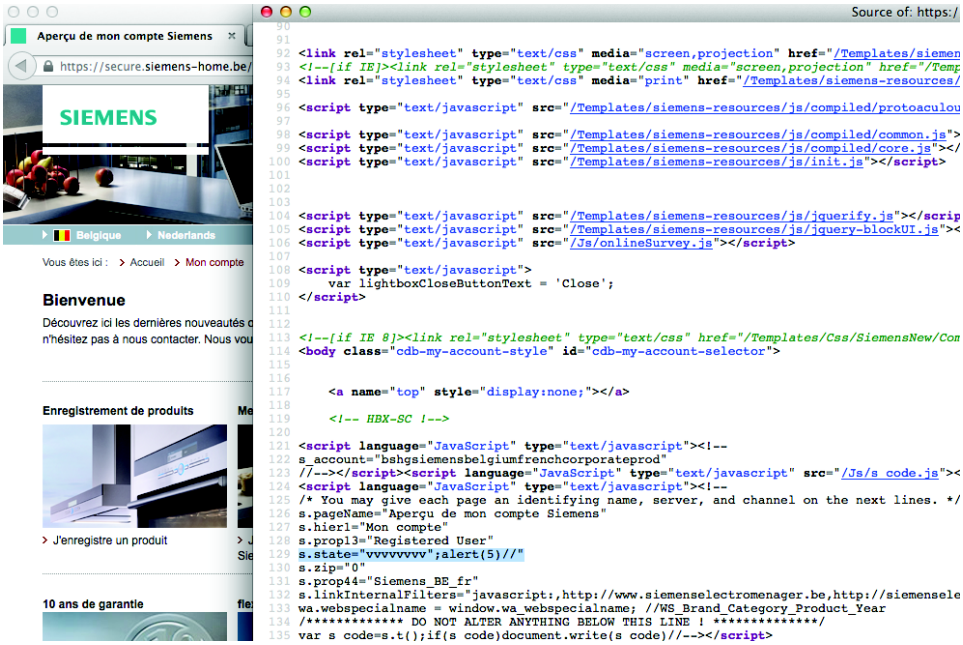


Figure C.11: Siemens–Type-2 Reflection in a JS context - code

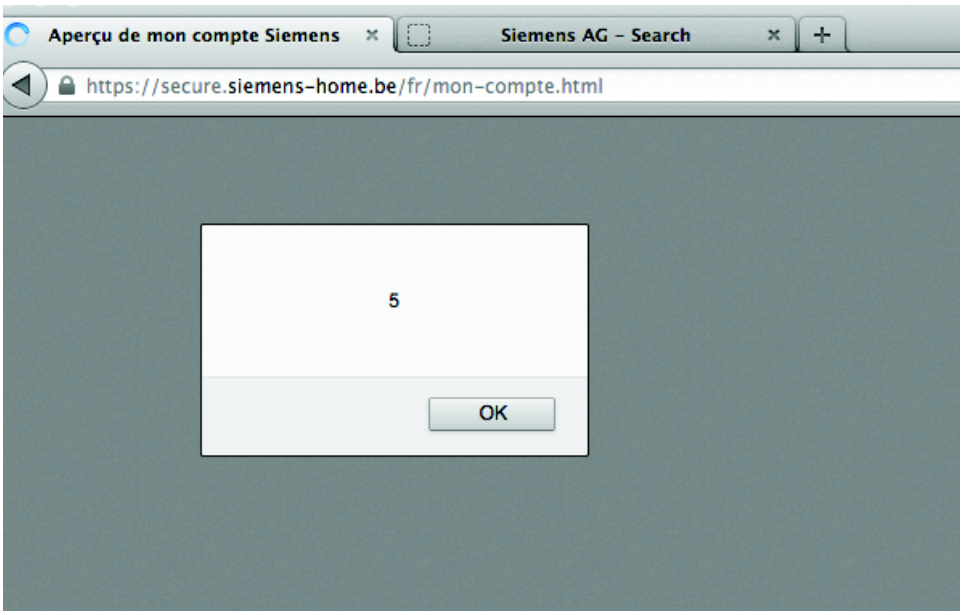


Figure C.12: Siemens–Type-2 Reflection in a JS context - execution

Two type-2 Reflections in HTML context result in IE-7 XSS This Type-2 XSS involves two parameters and I only succeeded in exploiting this reflection in one browser. We have two POST parameters in which we can inject characters: `name`

and `firstname` (see Figure C.13).

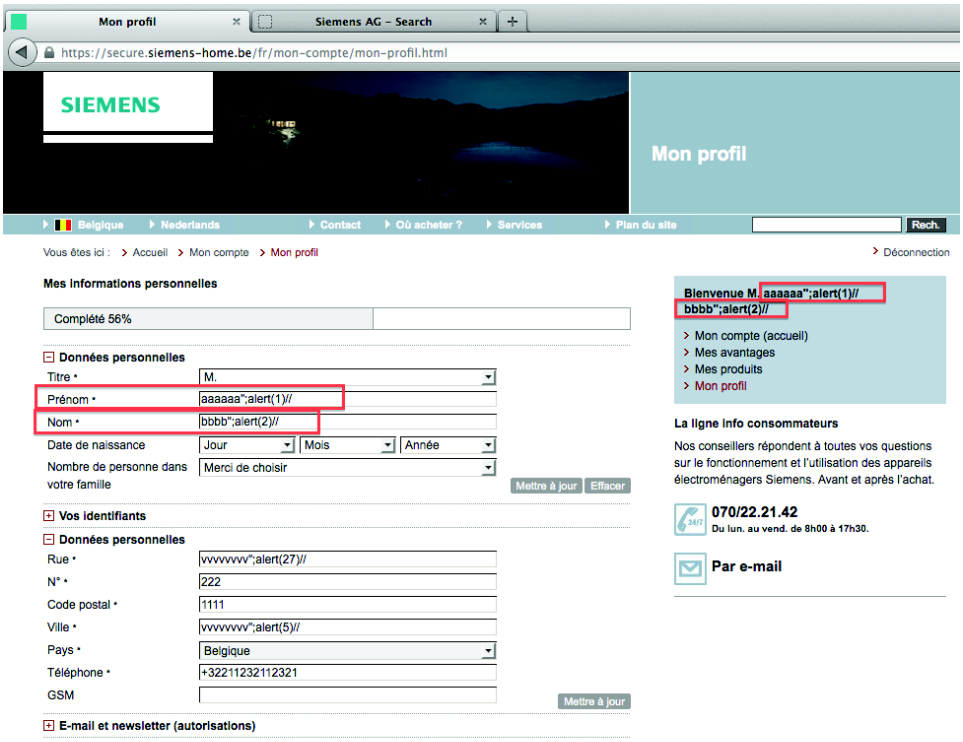


Figure C.13: Siemens–Two Parameters Confusion - Reflection

This is not a trivial XSS in the sense there is a server sanitizer in place (maybe a WAF). So far, its observed behavior exhibits the following constraint:

Constraint 1: If the substring `<` *immediately followed by a character in* `[a-Z]` *is present in any parameter, then the request is not processed, and we are redirected to an error page (see Figure C.14).* Thus the fuzzed values `<b` or `<u` are blocked by the sanitizer.

Figure C.14: Siemens–Two Parameters Confusion - Filter

Hopefully, as [Heiderich 2012b] pointed out: `<%` permits inserting a node in Internet Explorer 7.

Good starting point, we have injected an active DOM node.

Constraint 2: the sanitizer blocks input parameters which are more than 40 characters long.

However, on such a node, classic JS event handlers (e.g., onmouseover, etc.) cannot be triggered. We need another method for triggering a scripting engine. What about CSS? `:after` and `:before` only work starting from IE8, and the first constraint permits only IE7-10 node injection. Thus this first candidate solution is discarded.

Microsoft integrated in Internet Explorer 5.0 a feature named Dynamic Properties, aka CSS expressions [Microsoft a]. And those are interpreted in IE7-10 [Braun & Heiderich 2013].

Thus we attempt `name=<%` and `firstname= style='a:expression(alert(1337))'>`.

But now another subtlety of the sanitizer is revealed:

Constraint 3: [a-Z]:[a-Z] would be blocked from going into the database, thus not being reflected.

Hopefully, this part of the filter can be bypassed by only putting a space: the string `a: b` is accepted. Moreover, the filter does not remove simple quotes `'`.

Finally: `name=<%` and `firstname= style='a: expression(alert(1337))'>`

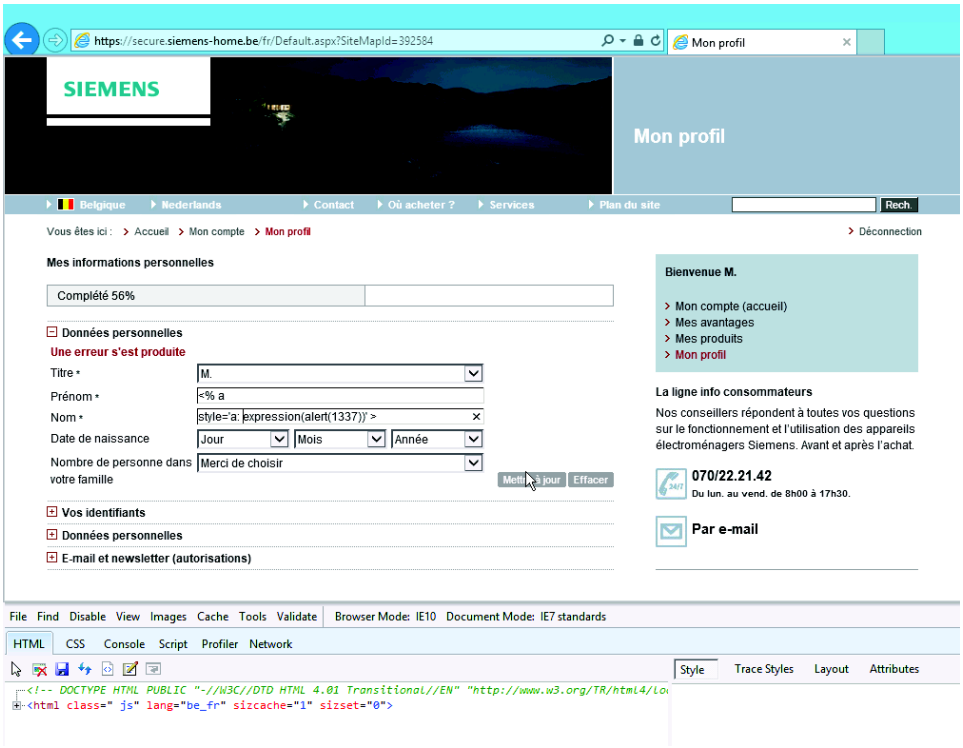


Figure C.15: Siemens–Two Parameters Confusion - Exploit IE7-IE10

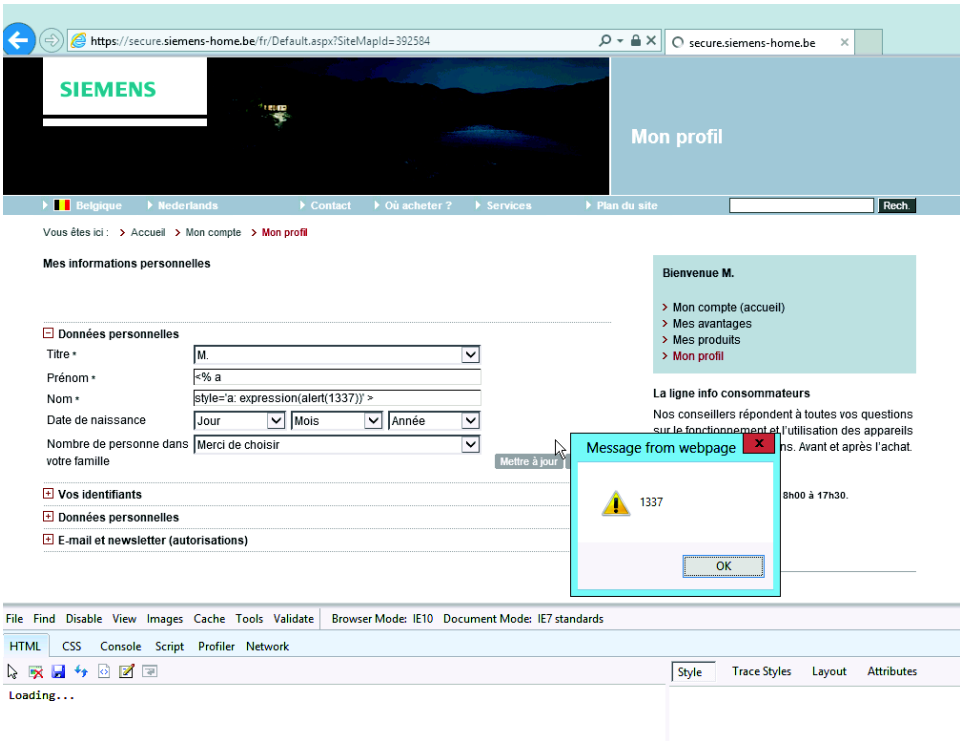


Figure C.16: Siemens–Two Parameters Confusion - Execution