FlashOver: Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications

Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, Frank Piessens
IBBT-Distrinet, Katholieke Universiteit Leuven, 3001 Leuven, Belgium
Steven.VanAcker@cs.kuleuven.be

ABSTRACT

Today's Internet is teeming with dynamic web applications visited by numerous Internet users. During their visits, typical Web users will unknowingly use tens of Rich Internet Applications like Flash banners or media players. For HTML-based web applications, it is well-known that Cross-site Scripting (XSS) vulnerabilities can be exploited to steal credentials or otherwise wreak havoc, and there is a lot of research into solving this problem. An aspect of this problem that seems to have been mostly overlooked by the academic community, is that XSS vulnerabilities also exist in Adobe Flash applications, and are actually easier to exploit because they do not require an enclosing HTML ecosystem.

In this paper we present FlashOver, a system to automatically scan Rich Internet Applications for XSS vulnerabilities by using a combination of static and dynamic code analysis that reports no false positives. FlashOver was used in a large-scale experiment to analyze Flash applications found on the top 1,000 Internet sites, exposing XSS vulnerabilities that could compromise 64 of those sites, of which six are in the top 50.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

Keywords

Flash, Rich Internet Applications, XSS, Cross-site Scripting Vulnerabilities, Automated Interaction, Large-scale Experiment

1. INTRODUCTION

The last fifteen years have transformed the Web in ways that would seem unimaginable to anyone of the "few" Internet users of the year 1995 [37]. What began as a simple set of protocols and mechanisms facilitating the exchange of static documents between remote computers is now an everyday part of billions' of users life, technical and non-technical alike. The sum of a user's daily experience is composed of open standards, such as HTML, JavaScript and Cascading Style Sheets as well as proprietary plugins, such as Adobe's Flash [4] and Microsoft's Silverlight [27].

Adobe's Flash is the most common way of delivering Rich Internet Applications to desktop users, with the latest statistics revealing an almost complete market penetration of Flash on desktop computers [13, 32]. While some have claimed that the new version of HTML, HTML5 [16], contains enough functionality to render the use of Flash obsolete, the reality is that today most Rich Internet Content, ranging from advertising banners and video players to interactive photo galleries and online games, is served and consumed by the Flash platform.

This rapid evolution of the Web was not left unnoticed by attackers. Traditionally, attackers preferred attacking the server-side of the Internet infrastructure, such as Web servers [21] and mail servers, since that gave them access to powerful hosts with plenty of bandwidth and disk space as well as a foothold in a company's internal network. Nowadays however, the attacks are targeting the client-side of the Internet infrastructure. This can be the Web application, as rendered in a browser, the software of the browser itself or even the user sitting behind the browser. The result of client-side attacks is usually the theft of user credentials or the download of malware that makes the user's computer an unwilling part of a botnet [12].

Since Flash is part of all the technologies that shape the every day experience of Web users, it is also part of this new attack surface. Attacks against Flash target either vulnerabilities in the code of the Flash platform itself, or the insecure practices of developers of Flash applications. In this second category falls the problem of Cross-site Scripting (XSS) [43]. While XSS in Web applications is a well-known and extensively researched problem, the problem of performing Cross-site Scripting attacks through vulnerable Flash applications has received much less attention. A Flash application can interact with the DOM (Document Object Model) of the page that embeds it or even with the browser itself. This allows Flash developers to read information from the page that embeds them, write information to the DOM or redirect the user to a desired page, such as the redirection that happens when a user clicks on a Flash advertisement banner. If these interactions are not protected adequately, an attacker can inject arbitrary JavaScript code that will be executed by a victim's browser in the context of the web-

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

site hosting the vulnerable Flash application. Such code can, among others, steal a user's session identifier, access the website's local storage on a victim's browser or, in some cases, read the victim's geolocation information.

In this paper we present FlashOver[1], a system capable of automated detection of Cross-site Scripting vulnerabilities in Flash applications. As the name of our system implies, its goal is to discover ways to perform malicious interactions between a Flash application and the rendering browser, that were never intended by the programmer of the vulnerable application. Given a Flash application, FlashOver performs static analysis in order to automatically identify ActionScript variables that can be initialized with user-input and are also used in operations that are commonly prone to code injection attacks. The identified variables are then tested dynamically in order to discover actual vulnerabilities present in the audited Flash application.

More specifically, our FlashOver prototype first decompiles the byte-code representation of ActionScript (the scripting language of the Flash platform) and then performs static analysis on the source code of the application, in search for commonly misused function calls that are responsible for *Flash-to-DOM* and *Flash-to-Browser* communication. Once these functions are located, our system then tracks the arguments of these function calls back to their initialization. When this process is complete, the static-analysis component FlashOver produces a list of variables which are utilized in commonly misused ActionScript API calls and are initialized using user-input. This list of *potentially exploitable variables* is then used by the dynamic-analysis component of our system, which renders the Flash application in the Firefox browser and initializes the variables in many possible ways, always mimicking the methodology of attackers who would lure victims in a page under their control. In the last phase, the automatic clicking module of FlashOver clicks thousands of times on the rendered application, with the intent of triggering the vulnerable API call. If our system detects the execution of the injected JavaScript, then the Flash application is flagged as vulnerable.

To evaluate FlashOver, we obtained a partial list of Flash applications hosted on the top 1,000 sites of the Internet, which we downloaded and provided as input to our system. At the end of the experiment, FlashOver successfully detected exploitable XSS vulnerabilities in Flash applications of many well-known websites, including `ebay.com`, `skype.com`, `mozilla.org` and `apple.com`. These results are evidence both of the problem of XSS attacks through Flash applications as well as our system's ability of automatically detecting them. The main contributions of this paper are the following:

- Detailed analysis of an XSS attack vector that is commonly overlooked in Web application development
- Design and implementation of FlashOver, a fully automated system which uses a combination of static and dynamic analysis in order to identify Flash applications vulnerable to code injection attacks
- Evaluation of our system using Flash applications of the top Internet websites, showing the prevalence of the aforementioned vulnerability as well as our system's ability of detecting it

[1]*flashover*: An unintended electric arc, as between two pieces of apparatus

The rest of this paper is structured as follows: In Section 2 we give a brief overview of Cross-site Scripting attacks, Flash technology and how the one affects the other. We describe the general architecture of FlashOver in Section 3 followed by our implementation choices and their rationale in Section 4. In Section 5 we evaluate our prototype by using it to discover previously unreported vulnerabilities in Flash applications of the top 1,000 Alexa sites. We present our ethical considerations in Section 6, we discuss related work in Section 7 and we conclude in Section 8.

2. BACKGROUND

In this section we give a brief overview of Cross-site Scripting attacks and of the Adobe Flash platform. We also present a motivating example showing how a vulnerable Flash application can be used to inject malicious JavaScript that will be executed by user's browser in the context of the domain hosting the vulnerable Flash application. While the techniques presented in the rest of this paper are specific to the Flash platform, they are, in principle, applicable to other similar content-delivering platforms, such as Microsoft Silverlight [28].

2.1 Cross-site Scripting

Cross-site Scripting (XSS) attacks belong to a broader range of attacks, collectively known as code injection attacks. In code injection attacks, the attacker inputs data that is later on perceived as code and executed by the running application.

In XSS attacks, an attacker adds malicious JavaScript code on a page of a vulnerable website that will be executed by a victim's browser when that vulnerable page is visited. Malicious JavaScript running in the victim's browser and in the context of the vulnerable website can access, among others, the session cookies of that website and transfer them to an attacker-controlled server. The attacker can then replay these sessions to the vulnerable website effectively authenticating himself as the victim. The injected JavaScript can also be used to alter the page's appearance to perform phishing or steal sensitive input as it is typed-in by the user.

2.2 Adobe Flash

Adobe Flash is a proprietary multimedia platform which is used to create Rich Internet Applications. To be able to run Flash applications on a desktop, a Flash player must be installed which takes the form of a browser plugin. According to the latest statistics, Adobe's Flash player is installed on more than 99% of desktops connected to the Internet [13, 32]. Over the years, the amount of functionality available to Flash applications has increased with each new version of the Flash player. Today, a Flash application can combine audio, video, images and other multimedia elements.

Flash applications are contained in SWF files (i.e. files with the .swf extension) which bundle multimedia elements together with byte-code-compiled ActionScript (AS) code. When loaded into the Flash player, the Flash application is rendered and, if present, the AS byte-code is interpreted and executed. ActionScript is a scripting language developed by Adobe which allows the programmer to handle events, design the interaction between multimedia elements and com-

municate with both the embedding browser and remote Web servers. The current version of ActionScript is ActionScript 3.0 with legacy support for prior versions.

```
<object type="application/x-shockwave-flash"
    data="myFlashMovie.swf" width="550"
    height="400">
  <param name="movie" value="myFlashMovie.swf" />
  <param name=FlashVars
      value="var1=Hello&var2=World" />
</object>
```

Figure 1: Embedding a SWF file using the object tag

2.3 Using SWF files

SWF files are typically embedded in HTML using the `<object>` or `<embed>` tags, but it is also possible to load an SWF file into the browser directly, without embedding it into HTML, either by requesting it as is from a browser's URL bar or providing it as the source argument to an `<iframe>` tag in an existing HTML page.

Flash, like many other technologies, allows for the provision of load-time input next to hard-coded values specified at compile-time and present in the resulting SWF file. For instance, YouTube videos are displayed on webpages that each embed the same Flash video player. Data specific to the displayed video-file is passed to the Flash player at load-time through variables embedded in the enclosing HTML page. Flash supports two methods of passing values to Flash objects:

- **FlashVars directive**: When embedding a SWF file using the `<object>` or `<embed>` tags, the FlashVars parameter can be used to pass values to specific variables. In Figure 1, FlashVars are utilized to initialize Flash's variables `var1` and `var2` to "Hello" and "World" respectively.

- **GET parameters**: A web developer can also utilize GET-parameters to pass arguments to a Flash application. For instance, when the URI: `http://example.com/myFlashMovie.swf?var1=Hello&var2=World` is invoked, the Flash application will initialize its internal variables `var1` and `var2` with their respective values. This method is usually overlooked by web developers who believe that the Flash application hosted on their page can only receive the parameters that they have hard-coded in the embedding HTML page and thus in many cases do not perform input validation within the Flash application itself.

2.4 Execution context of SWF files

In the previous section, we briefly examined the two ways that a SWF file can be loaded by a browser (using special HTML tags or a direct reference). While in both cases, the Flash Player loads the SWF file and starts executing it, there is a very important difference in the way that the two Flash applications interact with the surrounding page when the Flash applications requests the execution of JavaScript code from the browser.

The *allowScriptAccess* [2] runtime parameter arbitrates the access a Flash application has to the embedding page. There are three possible values: 'always', 'sameDomain' and 'never', with 'sameDomain' being the default. This value has the effect that access is only allowed when both the SWF application and the embedding page are from the same domain.

When an SWF file is embedded using the embed tag, and Flash requests the execution of JavaScript code from the browser, the code will execute within the origin of the embedding site, assuming a suitable value for the *allowScriptAccess* parameter. That is, if a SWF file hosted on the web server of `foo.com` is embedded in an HTML page on `bar.com`, the *origin* of the Flash-originating JavaScript is now `bar.com`. The origin is defined using the domain name, application layer protocol, and port number of the HTML document embedding the SWF.

If however, `bar.com` loads the SWF file of `foo.com` using an `<iframe>`, the browser creates an empty HTML page around the Flash application and any JavaScript initiated from the application will retain the origin of `foo.com`. Additionally, since the default value for *allowScriptAccess* is 'sameDomain', this means that the Flash application will be able to access data in the same origin as `foo.com`.

2.5 XSS in Flash

```
movie 'ad.swf' {
  button 42 {
    on (release) {
      getURL(_root.clickTag, '_blank');
    }
  }
}
```

Figure 2: ActionScript 2.0 source code of an example vulnerable Flash application

Consider a Flash advertising banner of which the ActionScript 2.0 source code is listed in Figure 2. The banner includes a button which, when clicked and released, triggers the execution of the `getURL()` function. The `getURL(url, target)` directs the browser to load a URL in the given target window. In this example, the URL is obtained from the variable `clickTag` in the global scope, and loaded into a new window (`_blank`). When used legitimately, the banner is located on `http://company.com/ad.swf` and is embedded on one of `company.com`'s web pages. The value of the `clickTag` variable is provided by the embedding page using the `FlashVars` directive and, in our example, suppose that it would redirect the clicking user to e.g. `http://company.com/new_product.html`.

As described in earlier sections, a SWF file can be directly referenced and any GET parameters will be provided to the Flash application itself, exactly as in the `FlashVars` case. Thus, if the banner was directly requested through `http://company.com/ad.swf?clickTag=http://www.evil.com`, the `clickTag` variable would now hold the value `http://www.evil.com` instead of the value intended by `company.com`. This behavior could be abused by attackers in order to send malicious requests with the correct Referrer header towards Web applications that use Referrer checking as a means of protection against CSRF attacks [34]. While this is defi-

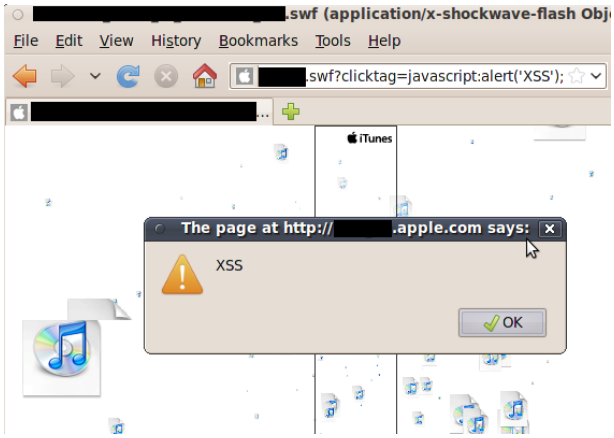

Figure 3: Advertising Banner on apple.com vulnerable to Cross-site Scripting through Flash

nitely a misuse scenario, the vulnerable code unfortunately allows for a much greater abuse. Instead of providing a website URL as the value for `clickTag`, an attacker could provide a JavaScript URL, such as `javascript:alert('XSS')`. A JavaScript URL is a URL that causes the browser to execute the specified JavaScript code in the context of the current-page (`alert('XSS')` in our aforementioned example) instead of making a remote request, as is the case in HTTP(S) URLs. In this scenario, when that banner is clicked, the user's browser will execute attacker-supplied JavaScript code instead of redirecting the user.

All an attacker needs to do in order to exploit this vulnerability, is to lure a victim into visiting a website which loads the vulnerable SWF file in an `iframe` and insert a `javascript:` URL containing malicious JavaScript code into the query string of the SWF file URL. Since the SWF file is loaded in an `iframe`, it will retain the origin of `company.com` and thus when the user clicks on the banner, the JavaScript code will execute in the context of `company.com` instead of the attacker's site. This will allow the malicious JavaScript code to access, among other things, the user's cookies for `company.com` and steal his session identifiers. If a click on the vulnerable Flash banner is required to trigger the execution of the injected JavaScript, the user can be tricked into clicking the banner, either using social engineering or clickjacking techniques [7]. In cases where the vulnerable code is triggered after a predetermined amount of time, all that the attacker needs to do is to make sure to keep the user on his malicious site for the appropriate amount of time.

While the example ActionScript in Figure 2 appears to be a contrived one, many websites unfortunately have similarly vulnerable banners. Figure 3 shows a banner hosted on `apple.com`[2] which does not perform input validation within its ActionScript code and is thus vulnerable to XSS.

3. FLASHOVER APPROACH

The goal of FlashOver is to automatically discover XSS vulnerabilities in Flash applications, as opposed to the manual code review illustrated in Section 2.5. Logically, FlashOver can be separated in three sequential steps: static analysis, attack URL construction and automated interaction. The high-level idea behind each of these steps of this approach is explained in more detail in the following subsections.

3.1 Static analysis

In this first step, *potentially exploitable variables* (PEVs) are automatically discovered in a given SWF file. PEVs are variables which are utilized in commonly misused ActionScript API calls and are initialized using user-input. This step requires a static analysis of the ActionScript byte-code embedded in the given SWF file.

Embedded ActionScript byte-code in an SWF file can not easily be read and understood by a human, giving a false sense of security to Flash developers who think their code can not be recovered. In reality, several free and commercial SWF decompilers exist that can reconstruct the ActionScript source code with very high accuracy.

Be it either through decompilation and source code analysis, or static analysis of the ActionScript byte-code, a list of potentially exploitable variables is extracted from the SWF file. The variables in this list will be used as attack vectors in later steps of FlashOver.

3.2 Attack URL construction

In this second step, an actual attack on the Flash application is prepared by crafting the *attack URL* that an attacker would give to a victim and trick him into navigating to it. In an actual XSS attack the attacker would try to execute JavaScript in the security context of a target domain using the victim's credentials for that domain. While the attacker's injected JavaScript would perform something undesirable for the victim, FlashOver uses the injected JavaScript code to log that the attack was successful.

The results of FlashOver will ultimately be used by Flash application developers to track down vulnerabilities in their code and fix them. Therefore it is essential that the results provide as much useful data as possible. There are three essential pieces of information that must be recorded to be able to reconstruct a successful attack: the *entry point* (i.e. Flash application that was exploited), the *attack vector* (i.e. the exploitable variable used to inject code) and the *payload* (i.e. the injected JavaScript code).

These three pieces of information are encoded in the attack URL. The SWF file being attacked can be identified by a unique identifier `swfid`. For each variable `var` of the potentially exploitable variables, as identified in the static analysis step, a payload value of payload-type `type` is generated. This payload contains JavaScript code that, when executed by the targeted Flash application, will log the tuple `(swfid, var, type)`. From any tuple `(swfid, var, type)` that shows up in the logs, the entry point, attack vector and payload can be reconstructed and can be used to identify the exact vulnerability of the Flash application.

3.3 Automated interaction

In the third step of the FlashOver process, the previously crafted attack URLs are used to truly attack the Flash application being examined. In a real-world scenario, the attacker would give the attack URL to a victim and trick the victim into interacting with the given Flash application. Since FlashOver tries to match the scenario as close to reality as possible, an automated process must interact with the

[2]We discovered this vulnerable SWF file through our experiment described in Section 5, and we also responsibly informed Apple about this vulnerability, see Section 5.3

Flash application and by doing so, trigger the execution of the JavaScript payload encoded in the attack URL.

Interaction can mean a lot of things. Flash applications can respond to keyboard events, mouse events and even more esoteric events from e.g. a built-in tilt sensor. The set of input events that trigger actions in a Flash application depends on the Flash application itself. For good results, the automated interaction process should try to cover as much as possible in an intelligent way.

4. FLASHOVER PROTOTYPE

The description of the general FlashOver approach in Section 3 omits implementation details, because each of the steps in FlashOver can be implemented in a number of ways with varying degrees of thoroughness. We purposefully chose to implement a minimalistic version of FlashOver to investigate the level of effort and skill required by an attacker to automatically detect XSS vulnerabilities in SWF files.

Our FlashOver prototype is schematically illustrated in Figure 4. The following subsections discuss the implementation details of each step in our FlashOver prototype.

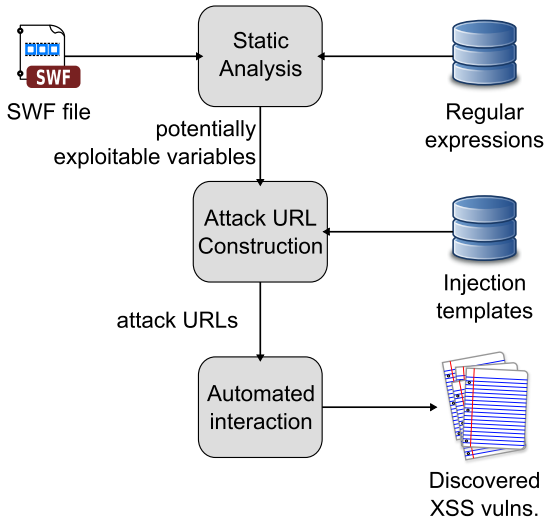

Figure 4: Schematic overview of our FlashOver prototype: During static analysis, the SWF file is decompiled and regular expressions uncover *potentially exploitable variables* (PEVs) from the ActionScript source-code. These PEVs are inserted into injection templates in the attack URL construction step. The attack URLs are loaded in a real browser in the automated interaction step, resulting in a list of discovered XSS vulnerabilities.

4.1 Static analysis

This first step in the FlashOver process requires static analysis of the SWF file. We chose to decompile the SWF file and then perform a simple static analysis on the resulting ActionScript source code.

There are many SWF decompilers, but not all of them support ActionScript 3.0. Choosing a decompiler, such as the freely available `flare` [24], that does not support the latest version of ActionScript, would mean that there would be a blind-spot in our analysis. For that reason, we chose a commercial decompiler with support for ActionScript 3.0 [35].

To reduce the complexity of our prototype, we opted for a simple regular-expression extraction of the PEVs instead of using more complicated analysis methods. Using this method, the resulting ActionScript source code is searched for patterns indicating potentially exploitable variables.

- `_root.`*re*
- `getRemote(#,`*re*`,...)`
- `.addCallback(#,#,`*re*`)`
- `.sendAndload(`*re*`,...)`
- `loadvariables(`*re*`,...)`
- `URLRequest(`*re*`,...)`
- `getURL(`*re*`,...)`
- `loadMovie(`*re*`,...)`
- `.load(`*re*`,...)`
- `.call(`*re*`,...)`
- `loadClip(`*re*`,...)`

where the regular expression to match a variable name *re* = '`[a-zA-Z$_][a-zA-Z0-9$_]*`' and '#' denotes a "don't care" parameter.

Figure 5: The regular expressions, in pseudo-form, used in our FlashOver prototype to match the names of potentially exploitable variables

The regular expressions used in our prototype are listed in pseudo-form in Figure 5. For each of these regular expressions, *re* indicates where the name of a potentially exploitable variable could appear in a function call in the ActionScript source code. The regular expression used to match variable names is synthesized from the variable naming rules defined by Adobe: *"The first character of an identifier must be a letter, underscore (_), or dollar sign ($). Each subsequent character can be a number, letter, underscore, or dollar sign"* [1]. The first regular expression (`_root.`*re*) indicates that a variable in the global address space is used, while the other regular expressions match function calls for sensitive functions that could lead to XSS.

4.2 Attack URL construction

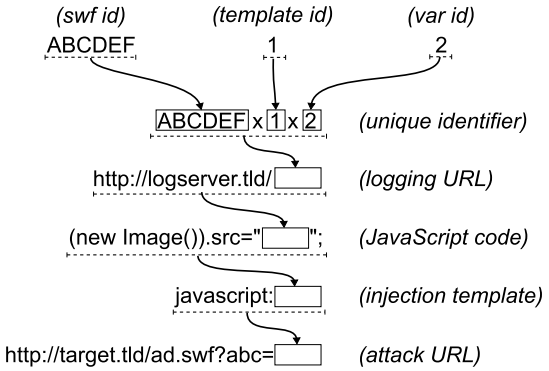

Figure 6: Construction process of an attack URL for `http://target.tld/ad.swf` with swfid `ABCDEF`, injection template id `1` and variable `abc` with id `2`

Based on the variable names identified in the previous step, attack URLs are constructed that, when the attack payload is triggered, will report in what way the given SWF file is vulnerable to XSS.

Exploitable variables can be used in ActionScript in a number of different ways. Through our review of JavaScript injection techniques, we identified a non-exhaustive list of

id	Example occurrence of *var*	Contents of *var*	
0	`getURL(`*var*`)`	*target URL*	control - plain target URL
1		`javascript:`*code*	JavaScript URL
2	`getURL("javascript:"+`*var*`)`	*code*	JavaScript code by itself
3	`writeHTML(`*var*`)`	`<script>`*code*`</script>`	HTML <script> tag injection
4	`eval("x = "+`*var*`+";")`	`0;` *code*`//`	introducing closing quotes and semicolons
5	`eval("x = '"+`*var*`+"';")`	`';` *code*`//`	
6	`eval("x = \""+`*var*`+"\";")`	`";` *code*`//`	
7	`eval("alert("+`*var*`+")")`	`0);` *code*`//`	introducing closing quotes, brackets and semicolons
8	`eval("alert('abc = '"+`*var*`+"')")`	`');` *code*`//`	
9	`eval("alert(\"abc = \""+`*var*`+"\")")`	`");` *code*`//`	

Figure 7: The 10 injection templates used in our implementation. Each injection template matches a certain example occurrence of a exploitable variable in ActionScript. The injection template indicates what data should be injected for a successful attack. The first template is a control, where the logging URL is injected instead of any code. The other nine inject actual JavaScript code.

nine ways in which an attacker-specified payload can ultimately be injected into a JavaScript context, through exploitable variables in an SWF file. As a control, we also use an injection template that injects no JavaScript code. The injection templates are summarized in Figure 7. For each of these injection templates, a separate attack URL is constructed.

As discussed in Section 3.2, the attack URL should encode information about entry point, attack vector and payload type into a unique identifier. The entry point is encoded by a unique hex-encoded 256-bit number that identifies the SWF file being analyzed. The attack vector, or the exploitable variable used to inject the payload, is encoded as an index into the list of identified potentially exploitable variables. Finally, the payload type is encoded as an index into the list of nine injection templates specified earlier.

The process for building an attack URL for an example SWF file with `swfid` equal to `ABCDEF`, an exploitable variable `abc` and injection template `1` is shown in Figure 6. From the given SWF file identifier (swfid), injection template index (type id) and exploitable variable index (var id), a unique identifier is constructed for this specific attack URL, by concatenating these three values, separated by a 'x' character. This unique identifier is appended to the URL for the log-server, forming the logging URL. The logging URL is then used in a JavaScript code fragment that, when executed, will trigger a request to the log-server, logging the unique identifier. This piece of JavaScript code is then inserted into the selected injection template, forming the payload of the attack URL, in this case a simple `javascript:` URL. Finally, the payload is assigned to the exploitable variable (`abc` in Figure 6) in a query string of the attack URL.

4.3 Automated interaction

The final step of FlashOver, involves passing the crafted attack URL to a simulated victim and let that victim interact with it, potentially triggering the execution of the injected JavaScript. Based on our personal experience and the analysis of many Flash applications, we make the assumption that most interactions with Flash applications are achieved through mouse clicks. For that reason, we only consider this type of interaction in our prototype implementation.

The Flash application is loaded into a real Firefox browser. The browser itself is started in `Xvfb`, a virtual frame-buffer X server [3] and the virtual mouse attached to this `Xvfb` session is controlled through the `xte` program [4]. The `Xvfb` server is set up to offer a virtual frame-buffer of 640x480 pixels with 24-bit color to any program running inside. Firefox, running inside `Xvfb` is started full-screen (so 640x480) in kiosk mode. This means that all toolbars and menus are removed, and undesirable functionality, like printing, is disabled.

Once Firefox has started and loaded the Flash application, a list with 10,000 random (x,y) locations is generated and passed to `xte`, which moves the mouse to those locations and issues a click. After these 10,000 clicks, the automated clicker pauses to give the Flash application time to process the input, which could involve executing the injected JavaScript payload.

If the execution of the injected JavaScript is triggered as a result of one or more mouse-clicks, this will be recorded in our logging server. The detection of the injected codes' execution effectively creates a new set of *actually exploitable variables* which is a subset of the original *potentially exploitable variables* set, as that was generated in the first stage of FlashOver. The entries of the logging server can then be used, as previously explained, to pinpoint the exact place in the Flash application and the specific attack vector that can be used for a XSS attack.

5. EVALUATION

We evaluated our FlashOver prototype with a large-scale experiment to determine how many SWF files vulnerable to XSS are hosted on the Alexa top 1,000 Internet sites [5].

5.1 Experimental setup

For each of the domains in the Alexa top 1k, a list of publicly exposed SWF files was retrieved from Altavista using the query "*site:domain.com filetype:swf*" where domain.com would be a domain in our experiment.

The SWF files discovered through these queries were downloaded onto a local web server. Although the experiment could have been conducted using the SWF hosted on their original locations, we feared that it might potentially harm the targeted site. In addition, storing the SWF locally improved performance by reducing the time it took to load the SWF file into the browser.

[3] `http://www.xfree86.org/4.0.1/Xvfb.1.html`
[4] `http://linux.die.net/man/1/xte`

After the non-SWF or otherwise invalid SWF files were removed from the set of downloaded files, they were processed by FlashOver. The static analysis and attack URL construction steps of FlashOver were performed on all SWF files in advance to reduce overhead for the entire experiment. The final step, using an automated clicker, was performed in parallel on 70 dual-core computers.

Because the automated clicker clicks on random positions on the Flash application, each run of the automated clicker can yield different results. To increase the odds that the payload in the attack URLs was triggered, the entire dataset was processed by the automated clickers 20 times. The total experiment ran for approximately five days, approximately six hours per run.

5.2 Results

From Altavista, 18,732 URLs were retrieved. After downloading, 3,800 SWF files did not contain a valid Flash application. Of the remaining 14,932 SWF files, 35 caused our decompiler to destabilize and crash. From the 14,897 SWF files that were decompiled successfully, 8,441 were determined to have exploitable variables. For each of these 8,441 SWF files, 10 attack URLs were generated: one for each injection template listed in Figure 7. The final generated dataset contained a list of 84,410 attack URLs. All of these were processed in parallel by the automated clickers.

After analysis of the log files, 523 SWF files were found to load content from an attacker-supplied URL (i.e. *URL injection*) and 286 SWF files allowed the execution of attacker-supplied JavaScript code. These 286 vulnerable SWF files can be traced back to 64 Alexa domains, of which six are in the top 50.

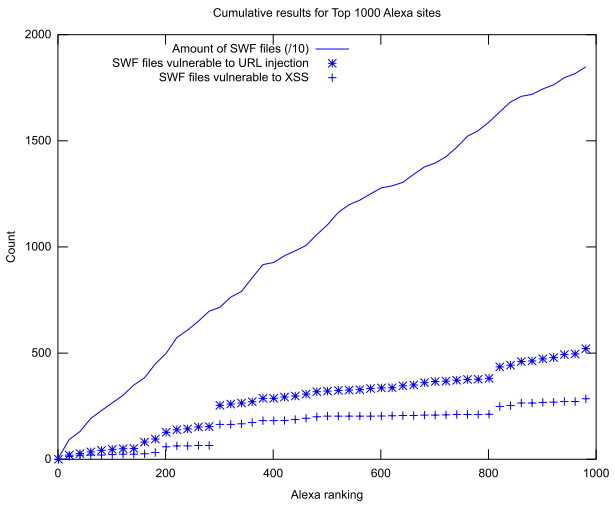

Figure 8: Results from our FlashOver experiment, shown as a cumulative plot. The amount of SWF files per site found is divided by 10 to match the scale of the other results.

The results of our large-scale experiment are summarized in the cumulative plot in Figure 8. The data-points are sorted on the x-axis, lower values indicating higher Alexa ranking, and vice versa. Three data-points per Alexa domain are shown: the amount of SWF files found per domain, divided by 10 to match scale, the amount of SWF files in that domain vulnerable to URL injection and the amount of SWF files vulnerable to XSS. The three distinguishable jumps, at indices 193, 293 and 806, indicate a large amount of vulnerable SWF files located at the Alexa domains of the corresponding ranking.

Variable Name	Instances found	Percentage
clicktag	101	35.31%
pageurl	97	33.92%
click	26	9.10%
counturl	10	3.50%
gameinfo	8	2.80%
link1	7	2.44%
url	3	1.05%
link04	2	0.70%
downloadaddress	2	0.70%

Figure 9: Top ten most commonly-named vulnerable variables found in our experiment

Figure 9 shows the ten most commonly named vulnerable variables that we discovered in our analysis. Interestingly, the two most commonly vulnerable variables are responsible for more than 69% of all vulnerabilities found. The fact that many different Flash applications are vulnerable to the same attack and through the same variables, suggests the use of automated tools for the creation of Flash applications that generate code in a vulnerable way. At the same time, our results highlight the need for scanning of variables and code-paths beyond the ones commonly associated with vulnerabilities.

5.3 Discussion

When one considers the number of vulnerable Flash applications found on the Internet's top websites, it becomes clear that XSS attacks through Flash applications are indeed a problem. Although Adobe advocates security best practices [3], stating that user-input should be sanitized where needed, this advice seems to be overlooked by Flash application developers.

The required effort and skill to automatically discover these XSS vulnerabilities is limited. As discussed in Section 4, our FlashOver prototype uses suboptimal static analysis and randomized clicking to simulate a user. For the static analysis part, a more precise taint-analysis system would produce better results since it could identify more variables influenced by user-input and thus produce a longer list of *potentially exploitable variables*. Moreover, a determined attacker can easily uncover additional vulnerabilities using a manual static analysis. Likewise, the randomized clicker is lacking the cognitive ability of an actual human user: it does not understand typical GUI widgets that a human would click and it can not interact with e.g. a game like a human would. This means that there may be vulnerabilities that our clickers couldn't trigger but that a human victim would. Therefore, the amount of vulnerable Flash applications detected in this experiment is a lower bound: the actual amount of vulnerable applications is most likely higher, making the security threat an even bigger issue.

An interesting property of FlashOver is that it detects successful JavaScript injection by actually simulating a victim who triggers the use of the injected JavaScript code in one or more potentially exploitable variables. Thus, while FlashOver may miss some vulnerabilities (false negatives), it

has practically zero false positives. While one can construct examples where FlashOver would report a false positive, e.g. an application that is vulnerable to XSS but inspects the injected payload and only allows it if it is "not dangerous", we believe that these are unrealistic examples and thus would not be encountered in the analysis of real-life Flash applications.

6. ETHICAL CONSIDERATIONS

Testing the security of real websites against Cross-site Scripting attacks may raise some ethical concerns. However, analogous to the real-world experiments conducted by Jakobsson et al. [18, 19] and Nikiforakis et al. [29], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real world. Moreover, we believe that our experiments will help raise awareness against this, usually overlooked, issue. In particular, note that:

- All Flash applications were downloaded and exploited locally thus no malicious traffic was sent towards the live Web servers of each website
- All attacks were targeting our own simulated victim and no real users
- We are in the process of disclosing these vulnerabilities to all the affected websites so that they may repair them

7. RELATED WORK

Due to the large installation percentage of Adobe's Flash in desktop and laptop computers, Flash has been the target of many attacks over the years. These attacks have been targeting either implementation bugs in the Flash plugin itself [10] or the insecure use of Flash functionality from Rich-Internet Application developers.

Cross-site Scripting attacks in Web applications [43] have received a lot of attention over the last years and there exists a wide range of research on detecting injected JavaScript and protecting the user from it [23, 40, 30, 39] as well as many initiatives that try to educate developers about this issue [31, 11]. The sheer volume of XSS attacks has even caused mainstream browsers like Microsoft Internet Explorer 8 and Google Chrome to add XSS-detection mechanisms in an attempt to stop attacks against the browsing user, even if the visited Web application isn't actively protecting itself [8, 33].

The problem of performing Cross-site Scripting attacks through insecure Flash API methods was first highlighted by Jagdale [17] who provided examples of insecure ActionScript code and reported that out of the first 200 SWF files that Google gave as a result to the search query "`filetype: swf inurl:clickTag`", 120 were vulnerable. Jagdale also showed that many tools that automatically generated SWFs were, at the time, generating applications vulnerable to XSS attacks, including tools by Adobe itself. Bailey [6] verified the earlier findings of Jagdale and gave examples of high-profile websites hosting SWFs vulnerable to Remote File Inclusion attacks (RFI) that could be leveraged to perform, among others, XSS attacks.

SWFScan [15] is a tool that decompiles a Flash application and performs static analysis to detect possible vulnerabilities. SWFScan searches a decompiled Flash application for hardcoded URLs, passwords, insecure cross-domain permissions and coding practices that may lead to XSS. SWFIntruder [36] is a user-guided semi-automatic tool which tests for XSS vulnerabilities in Flash applications.

The important difference that separates FlashOver from earlier work is that earlier work depended either on the manual or semi-automatic analysis of SWF files. Contrastingly, FlashOver is the first system that is able to discover "zero-day" vulnerabilities in a completely automatic fashion without relying on naming conventions of commonly vulnerable variables or user guidance. While FlashOver, due to its incomplete static analysis, may miss some vulnerabilities (false-negatives), it produces no false-positives since any reported vulnerability could only have been reported because that vulnerability was exploited.

Another problem that has attracted attention from the security community is the existence of insecure cross-domain Flash policies. The Flash plugin is able to conduct Cross-Domain requests in a way that violates the Same-Origin policy that exists in JavaScript. In order to overcome this problem, any website that wants to be contacted through Flash, must opt-in by placing a cross-domain policy file in its root directory that specifies which domains can be accessed and in what ways. Three recent independent studies [25, 20, 26] all discovered that a great number of websites deploy insecure cross-domain policies in a way that allows their users to fall victims to impersonation attacks, simply by browsing to a malicious website.

An interesting observation is that over the last few years, many researchers have shifted their focus and have designed and implemented a number of *blackbox* and *whitebox* systems that, like FlashOver, attempt to automatically detect vulnerabilities in Web applications. These systems are usually less precise than human analysts but can process data much faster and can track dependencies among hundreds of files. Balduzzi et al. [7] presented a system that automatically discovers clickjacking attacks through an instrumented Firefox browser and a series of plugins that detect the overlay of many objects at specifics coordinates within a Web page. NoTamper, by Bisht et al. [9], detects vulnerabilities that would allow a user to successfully perform HTTP parameter-tampering. Ford et al. [14] propose OdoSwiff, a system to detect deliberately malicious Flash ads through a combination of static and dynamic analysis.

Jovanovic et al. [22], Xie et al. [42] and Wassermann et al. [41] use static analysis on a Web page's source code in an effort to identify potential flaws that could lead to XSS, SQL injections and command injection attacks. Sun et al. [38] use static analysis to infer the intended access-control of Web applications and use their models to detect access control errors.

8. CONCLUSION

The constant innovation in the World Wide Web has allowed developers to use more and more the browser as the platform of choice for delivering content-rich applications to users. In this picture, the Flash platform by Adobe plays a very important role and is widely used in modern websites. However, since Adobe is a Web technology, it is also part of the modern attack surface where the targets are now the users and their browsers. In this paper, we analyzed the implications of making the wrong assumptions in the Flash platform and we presented FlashOver, the first fully auto-

mated discovery system for XSS attacks, specific to Flash. FlashOver uses a combination of static and dynamic analysis to identify vulnerabilities in real-life Flash objects and using our system, we discovered that a significant number of high-valued websites host Flash applications that are vulnerable to Cross-Site Scripting. These results attest towards the importance of this attack vector and we hope that our work will help raise awareness of insecure coding practices in the community of Rich Internet Application developers.

Acknowledgments: We would like to thank our shepherd, Dieter Gollmann, and the anonymous reviewers for their insightful comments that helped to greatly improve the presentation of this paper. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the IBBT, the Research Fund K.U.Leuven, the B-CCENTRE and the EU-funded FP7 projects NESSoS and WebSand.

9. REFERENCES

[1] Adobe. About naming variables. http://help.adobe.com/en_US/AS2LCR/Flash_10.0/help.html?content=00000047.html.

[2] Adobe. ActionScript 3.0 - Controlling access to scripts in a host web page. http://livedocs.adobe.com/flex/3/html/help.html?content=05B_Security_14.html.

[3] Adobe. Creating more secure SWF web applications. https://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html.

[4] Flash Player | Adobe Flash Player 11 | Overview. http://www.adobe.com/products/flashplayer.html.

[5] Alexa - Top Internet Sites. http://www.alexa.com/topsites.

[6] M. Bailey. Neat, new, and ridiculous flash hacks. In *BlackHat DC*, 2010.

[7] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, 2010.

[8] A. Barth. Chromium Blog: Security in Depth: New Security Features. http://blog.chromium.org/2010/01/security-in-depth-new-security-features.html.

[9] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 607–618, New York, NY, USA, 2010. ACM.

[10] D. Blazakis. Interpreter exploitation. In *Proceedings of the 4th Usenix Workshop on Offensive Technologies (WOOT)*, 2010.

[11] W. A. S. Consortium. Web Hacking Incident Database. http://projects.webappsec.org/Web-Hacking-Incident-Database.

[12] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Pc penetration | statistics | adobe flash platform runtimes. http://www.adobe.com/products/flashplatformruntimes/statistics.html.

[14] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious flash advertisements. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 363–372, Washington, DC, USA, 2009. IEEE Computer Society.

[15] Hewlett-Packard Development Company. SWFScan. http://h30499.www3.hp.com/t5/Following-the-White-Rabbit/SWFScan-FREE-Flash-decompiler/bc-p/5442703?jumpid=reg_r1002_usen.

[16] HTML5. http://dev.w3.org/html5/spec/Overview.html.

[17] P. Jagdale. Blinded by flash: Widespread security risks flash developers donâĂŹt see. In *BlackHat DC*, 2009.

[18] M. Jakobsson, P. Finn, and N. Johnson. Why and How to Perform Fraud Experiments. *Security & Privacy, IEEE*, 6(2):66–68, March-April 2008.

[19] M. Jakobsson and J. Ratkiewicz. Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In *15th International Conference on World Wide Web (WWW)*, 2006.

[20] D. Jang, A. Venkataraman, G. M. Swaka, and H. Shacham. Analyzing the Cross-domain Policies of Flash Applications. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[21] JoMo-kun. m0j0.j0j0 Guide to IIS Hacking. http://www.foofus.net/~jmk/iis.html.

[22] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[23] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.

[24] I. Kogan. no|wrap.be - flare. http://www.nowrap.de/flare.html.

[25] G. Kontaxis, D. Antoniades, I. Polakis, and E. P. Markatos. An empirical study on the security of cross-domain policies in rich internet applications. In *Proceedings of the 4th European Workshop on Systems Security (EUROSEC)*, 2011.

[26] S. Lekies, M. Johns, and W. Tighzert. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[27] Microsoft Silverlight. http://www.microsoft.com/silverlight/.

[28] Microsoft. Security in Silverlight.

http://msdn.microsoft.com/en-us/library/cc972657(v=vs.95).aspx.

[29] N. Nikiforakis, M. Balduzzi, S. Van Acker, W. Joosen, and D. Balzarotti. Exposing the lack of privacy in file hosting services. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, LEET'11, Berkeley, CA, USA, 2011. USENIX Association.

[30] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2011.

[31] OWASP Top 10 Web Application Security Risks. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[32] Rich internet application (ria) market share. http://www.statowl.com/custom_ria_market_penetration.php.

[33] D. Ross. Ie8 security part iv: The xss filter. http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx.

[34] C. Shiflett. Cross-Site Request Forgeries. http://shiflett.org/articles/cross-site-request-forgeries.

[35] Sothink swf decompiler. http://www.sothink.com/product/flashdecompiler/.

[36] Stefano Di Paola. SWFIntruder. http://code.google.com/p/swfintruder/.

[37] C. Stoll. The internet? bah! http://www.thedailybeast.com/newsweek/1995/02/26/the-internet-bah.html, 1995.

[38] F. Sun, L. Xu, , and Z. Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th Usenix Security Symposium*, 2011.

[39] M. Van Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, Feb. 2009.

[40] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS '07)*, 2007.

[41] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.

[42] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[43] The Cross-site Scripting FAQ. http://www.cgisecurity.com/xss-faq.html.