SWAN WG - Year 2011 Activity Report

Gregory BLANC (gregory@is.naist.jp)
Youki KADOBAYASHI (youki-k@is.naist.jp)

2011/12/31

1 Introduction

The SWAN WG carries out research in the field of Web 2.0 application security. SWAN stands for Security for Web 2.0 ApplicatioN. It was founded and started its activities in June 2010. It aims to bring forward Web 2.0 application security issues in the WIDE project as these issues are becoming more and more critical to services offered on top of the Internet. As a matter of fact, the IETF has also witnessed the recent foundation of the WebSec WG this year and SWAN actually preempted this creation.

1.1 What is Web 2.0?

There is no strict difference between Web 1.0 and Web 2.0 but it is universally understood that Web 1.0 applications rely mainly on the HTTP protocol to download pages in a synchronous pattern. On the other hand, Web 2.0 applications do involve abundant processing on the client side through embedded scripts transferring data to the server, even asynchronously, without the user experiencing delays. Web 2.0 does not rely on any particularly recent technology but on technologies that have been spreading the Web since its early years. JavaScript and XML, at the origin of the coined word Ajax (Asynchronous JavaScript with XML, 2005)[15], were technologies designed in the mid-1990 s. However what characterize Web 2.0 applications are their content-richness, their collaboration features (user participation, folksonomies, social networks), their ability to syndicate contents (aggregate sites, feeds, mashups) as well as their extensive use of the Ajax framework to perform dynamic, asynchronous HTTP transactions. Other essential objects comprise the Document Object Model (DOM), which is usually modified dynamically to avoid reloading web pages, JS Object Notation (JSON) objects, which are used for the serialization of structured data during XmlHTTPRequest (XHR) object transactions. Further information can be found in [19].

1.2 Web 2.0 Security Issues

For many years, web applications have been threatened by attacks such as SQL injections, directory traversals, buffer overflows, command injections and the likes. Then, the increasing popularity of BBS promote stored cross-site scripting (XSS) among attackers. Classic attacks usually attempt to abuse the targeted web server in order to take control of it. On the contrary, new generation attacks concentrate follow the paradigm shift prompted by Web 2.0 applications which are user-centric, cross-domain and rely on dynamic scripting technologies. As a matter of fact, new generation attacks often leverage scripting languages and trigger cross-domain bypasses to harm the user. Not to mention that Web 2.0 applications also often push much of the application logic to the browser allowing attackers to test their security as a whitebox. And vulnerabilities are not only found in the targeted application but also in script libraries, APIs or plugins the application uses, or in contents the application mashes from external origins.

Attackers have therefore learnt how to take advantage of such security holes to make their attacks stealthier and more massive. These attacks can leverage the user's browser to carry out a number of purposes ranging from information leakage, live keylogging, session riding or more elaborated schemes such as internal network fingerprinting, worm propagation or botnet management. Such sophisticated attacks are carried out through complex scripts hidden through layers of redirection and obfuscation and commonly known as malicious JavaScript or JS malware[23]. More information on Web 2.0 vulnerabilities and related attacks can be found in [8, 18]

1.3 Research Approaches

The SWAN WG is basically approach-agnostic in that it allows anyone to join and to propose its own approach to contribute to the fight against the proliferation of web-based attacks. Of course, ongoing projects do implement one specific approach

but this does not constrain all members to follow the same path; parallel, complementary or concurrent approaches are also welcomed. Indeed, there are many ways to deal with security issues in Web 2.0 applications either from the client or server's viewpoint or through the collaboration of the two sides. However it is usually agreed that server-side countermeasures are ineffective when attacks target users unless one is able to protect every web application on the Internet. Therefore, recent research works have been carried out on the client-side exclusively, except for secure mashup schemes[21, 25].

Some researchers have concentrated on the browser itself where many vulnerabilities lie and where exploitation takes place: initiatives aiming to sandbox the browser[16] or to enforce security policies in the browser[22] have been proposed. Other works focused on how to prevent some kind of attacks such as XSS[20] or CSRF[24] by designing some heuristics to characterize these attacks. To a much more granular level, JavaScript being the defacto standard in AJAX and in other programming frameworks, attackers have naturally took advantage of it being enabled in the victims' browsers. Consequently, some researchers took interest in how to mitigate such attacks by trying to minimize the impact of the JavaScript language: secure subsets[29], interposition[35, 39] or other hardening techniques have been elaborated. An adverse approach is not to restrain JavaScript in any way but rather to analyze programs to detect any malicious behavior: VM-based execution[31, 33], data tainting to prevent XSS[38] or even control flow analysis[17].

Though, we have favorized the latter approach in some of our works, we do not restrain WG members from contributing using any other approach.

2 Second Year Activities

From its inception, the SWAN WG has decided to concentrate on a rather precise objective in order to appeal to the community and launch projects as fast as possible. We launched a first project, the Web2Sec Testbed, a testbed to accommodate large-scale practical Web 2.0 application security-related experiments. The project does not only intend to build a practical testbed on top of the WIDE cloud but also to develop tools to be used within the testbed. While building a testbed seems relatively easy given the efforts deployed in other WIDE WGs such as WIDE-Cloud or ds1, the WebSec Testbed is yet to be constructed. However, the SWAN WG has been concentrating on developing analysis tools, especially to analyze JS malware which is a hot topic in Web 2.0 security.

For that purpose, we designed and started developing a proxy-based solution to analyze JS malware. This led to several publications on how to detect and extract obfuscation in JS malware[3] and how to automate deobfuscation to provide static analysis of JS malware[4, 5].

3 Web2Sec Testbed

The testbed is an all-purpose tool in that it allows designing large-scale experiments on both the offensive and the defensive side. By collaborating with the WIDE-cloud WG, we propose to deploy a testbed to perform experiments on Web 2.0 security issues. Similar to previous works from Stanford University[7], the deployment is two-fold. We propose on one end cloud services that will simulate a vulnerable and/or hostile Web 2.0 environment and on the other end, users will be provided with virtual machine (VM) images comprising several browsers (and several personalities) equipped with some plugins for defense and analysis, as well as several tools to perform security audit or attacks.

3.1 Motivation

The main motivation for such testbed is the one for any testbed: we cannot perform large-scale experiments in the wild and need to recreate a practical environment to provide containment of our experiments. Furthermore, large-scale attacks on social networks have already been witnessed and it is legitimate to try to understand propagation schemes better in order to mitigate these. But this cannot be reproduced on a small network, thereby the need for a larger experiment environment.

3.2 Overview

Figure 1 shows a simple overview of the tools we will deploy and develop for the Web2Sec testbed. As stated previously, the deployment is two-fold. On the server-side are present Web services we will provide to create the experiment environment and on the client-side are listed tools that will be included in the VM image: aside from browsers, proxies, vulnerability scanners, automation engines, attack frameworks will also be featured. On the right side of Figure 1, we present some tools we wish to develop in order to support experiments carried out in the testbed. At least 3 tools are to be engineered: a web security scanner for Web 2.0 applications, an AJAX monitor to track AJAX transactions generated by the browser and a JavaScript debugger to

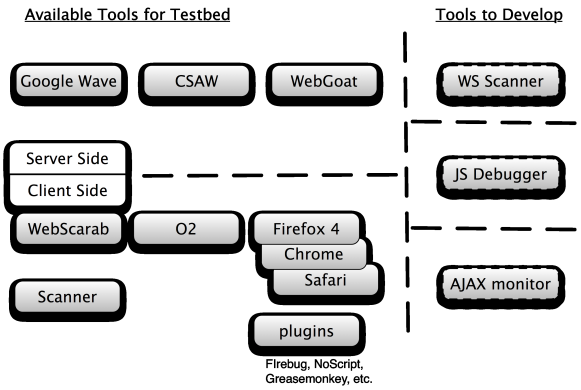

Figure 1: Overview of the tools to be deployed in the Web2Sec testbed

analyze JS programs and their behavior.

3.3 Applications

The Web2Sec testbed will support both offensive and defensive experiments in the realm of the Web 2.0. It can be used to perform sophisticated web application security evaluations and measure their impact on different browser personalities. We can also evaluate independently the security of different browsers against different type of attacks. It can also be used to foresee next-generation attacks and monitor the propagation of Web 2.0 attack vectors. Instrumentation of browsers and applications can also provide several interesting measures. Not to mention that such tool can also be used for education purposes as it was done in previous research works cited in Section 3.4

3.4 Related Works

Though testbed is not a recent research topic, there have not been any proposal on web security oriented testbeds until 2010. Contributions are not revolutionary but rather attempt to shed light on the critical issues the web community incurs. Unsurprisingly, most of the proposed testbeds were designed and used for educational purpose, the most outstanding being the Webseclab[7] used in web security classes in both Stanford and CMU. Our testbed basically follows the same design with on one end the cloud service and on the other end VM images. Webseclab's cloud service does actually perform class administration tasks while every student gets a VM distribution that contains the class exercises and tools to resolve them. Connecting to the cloud service allows for rating, downloading class materials and updating contents. Other notable testbeds have been the Blunderdome[27] which is rather an academic multi-layer service offensive security testbed simulating a university network and moth[6], a VM image that contains a set of vulnerable web applications and scripts for testing web application security scanners or static code analysis tools.

4 JavaScript Analysis Proxy

One of the other main projects of the SWAN WG is the development of a JS analysis proxy.

4.1 Motivation and Approach

Web 2.0 applications make an important use of the JavaScript (JS) language through the AJAX framework and it has proven to be a critical component of modern applications in terms of security, to the extent that is it often considered safer for users to disable JS in the browser. However, there is an important shortfall regarding user-experience and some popular applications are simply not accessible without JS enabled, not to mention the *addiction* of users to eye-candy interfaces.

Therefore, it has become obvious that such users are in need of systems able to provide them with a **safe** and **usable** Web experience. Earlier works have already tackled how to prevent attacks either on the server-side or on the client-side but often rely on heuristics an attacker can mimick. Lately, it has been understood that Web 2.0 security issues are more and more exploited through client-side vulnerabilities, notably by abusing the Same-Origin Policy (SOP) weakness and injecting remote JS payloads through Cross-Site Scripting (XSS) vulnerabilities. Attack payloads are often obfuscated and infected pages use many anti-analysis techniques, making straightforwared dynamic analysis techniques somehow difficult to use. Many proposals have also focused on analysis but fail to provide a consistent context, especially in context-dependent obfuscations against which forensic analysis is useless.

Additionally, a recent technical report from Google[34] pointed out several weaknesses of current web malware detectors. In particular, it distinguishes four prevalent classes of detectors: virtual machine-based detectors (web honeypots), emulation-based detectors which are the successors of virtual machine-based detectors, reputation-based detectors (blacklists) and signature-based detectors (antiviruses). The authors of this report surveyed circumvention techniques in modern web malware:

- social engineering are a set of techniques that require the end-user interaction in order to circumvent passive web honeypots;
- cloaking are a set of techniques that prevent analysis from emulation-based environments by detecting their presence;
- redirection are a set of techniques to hide malware distribution websites from blacklists by leveraging a redirection network made of numerous domains redirecting to each other;
- obfuscation are a set of techniques that render the code of program unintelligible to a human user and thwart signature-matching.

The authors propose to combine existing approaches in order to increase effectiveness.

On the other hand, we remark that actual web malware detectors that propose to analyze JavaScript are often execution-based which lead to undesired effects. First, the side-effects of execution have prompted attackers to design anti-analysis techniques to thwart dynamic analysis such as cloaking and obfuscation. Second, executing a malware can result in exploitation if the execution is not properly contained, prompting an offline analysis, that is analysis does not take place during browsing the Web. This has the consequence to render offline analyzers less attractive to end-users due to a low usability. Indeed, it is difficult to coerce users into analyzing scripts on a web malware analysis website each time they suspect a page of containing malware.

We also remark obfuscation is not really considered an issue by past research: either it is seen as an indicator of malice[9, 26] (while it is not[33]) or it is seen as trivially deobfuscated during execution[11, 36, 12]. However, recent advances in cloaking have demonstrated a hardening of obfuscation techniques that are able to stop deobfuscation in the case an analysis environment is detected. Often, benign contents are produced in order to confuse analysis.

We propose to apply a static, that is execution-less, approach to JS malware. The above remarks indicate that such analysis should take place on unobfuscated malware in order to be successful. Since static examination of obfuscated JS code cannot yield any result, it is necessary that we design a method to reliably cancel obfuscation while preventing execution of the JS code.

4.2 Threat Model

A common scenario unfolds as follows (cf. Fig. 2):

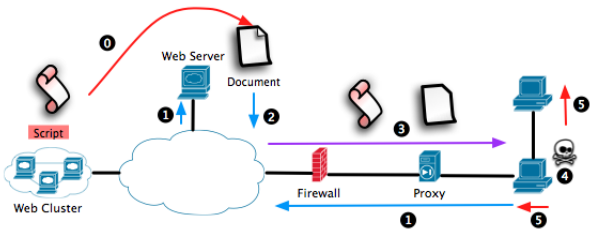

Figure 2: Typical Web infection threat model scenario

- step 1: the victim browses an infected web application (infected at step 0);
- step 2: the server will process requests from the user and may include infected contents in its response;
- step 3: the malware content possesses some specificities that makes it possible to bypass deployed security devices;
- step 4: the browser parses the web contents and eventually executes embedded scripts or download linked scripts that will be then executed. The user can also be trapped into a redirection or have a malicious iframe injected into the page she is browsing;
- step 5: upon execution, the script does something harmful and might communicate results back to an attacker's remote server or simply take silent actions that will profit the attacker or else make the victim download some malware;
- in most advanced scenarios, several attacks are combined to produce more massive harm leveraging either the participatory or social characteristics of Web 2.0 applications, or the loosely secured internal network of the victim.

One may opt that such scenarios do seldom occur. However, isolated infected users might not notice they are part of a massive attack, and companies often fail (sometimes on purpose) to report on such events. Hence, the publicity of large-scale Web 2.0 attacks remain lower than reality.

4.3 Overview

JS malware are stealthy, polymorphic and highly use obfuscation to conceal its malicious intent. It is often hidden through several layers of redirection and obfuscation. Since, we cannot rule that an obfuscated script is malicious[33], we need to

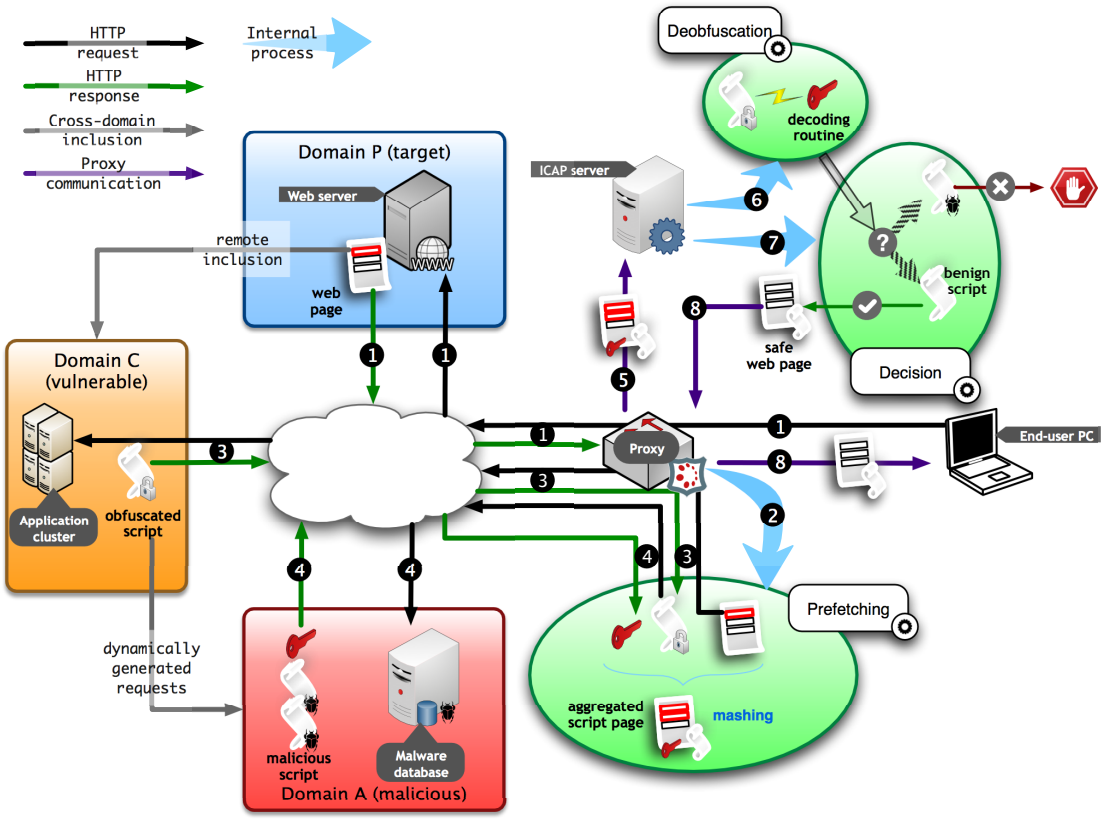

Figure 3: Overview of the Proposed Solution

deobfuscate it to ensure a sounder and more complete analysis. Besides, we require static analysis methods in order to maximize code coverage in a single pass. However to fight against different anti-analysis tricks specific to obfuscation schemes, we need to convey enough usable context to the analyzer. This can be done by parsing suspicious linked contents and fetching potential deciphering scripts. In order to tackle client-side JavaScript, we decided to deploy our solution on the client-side and to avoid putting any trust on the server-side since it is likely to be infected. However, since we need a consistent context in order to conduct JS program analysis, a realtime processing becomes necessary, which implies new constraints on our design solution. On one hand, program analysis can be a computation-intensive task and we should not impose it on the browser which is already busy with Web 2.0 application rendering. Additionally, if we were to partly execute suspicious code, we would rather avoid the browser performing such a risky task. On the other hand, we are looking for a realistic browser context in order to analyse the JS program as precisely as possible, but since we are aiming at performing static analysis, the context of the application and the personality of the browser can be provided to a proxy.

For these reasons, we advocate the use of a proxy-based solution, we named (sak_mis) which would sustain the load of realtime static analysis on JS candidates. The system is designed to work as follows (please refer to Fig. 3):

1. (sak_mis) is a proxy that intercepts HTTP requests issued by the end user and subsequent responses returned by the server;

2. upon reception of an HTTP response, the proxy kicks off the *prefetching* stage:

3. the requested web page is parsed to detect script inclusions, links and potential malicious locations (iframes, images, etc.). Script contents are then retrieved (prefetching) and inlined into the original web page;

4. if newly downloaded contents also contain links or inclusions to contents of interest, prefetching is also performed on these contents. This scenario also applies when new inclusions are uncovered after deobfuscation;

5. once prefetching is completed, the aggregated script page is sent to an external application server;

6. the application server is responsible for automating *deobfuscation*: obfuscated contents and decoding routines are extracted first. The deobfuscation stage of the attack scenario is emulated by the server. The process is repeated in case the deobfuscation yielded obfuscated contents as well;

7. once scripting contents can be directly interpreted by the machine, the *decision* module applies static analysis to extract a model of the script's intents. This model is compared to a knowledge base in order to infer whether it is a model of malicious intents or not;

8. in case the script is benign, the deobfuscated script is injected back into the original web page and served to the end user.

Since the goal is ultimately to decide on the malice of a given JS program, the proposed solution should prepare the JS candidate to be analyzed in the most efficient way. To do so, we need to deobfuscate obfuscated contents in order to provide readable code from which we can compute abstract semantics. But again, to achieve such task, we should ensure enough code is actually available to reverse the cipher (in case of a cipher obfuscation). Obfuscation techniques actually rely on several layers of links and redirections, as well as layers of obfuscation, therefore we need to ensure that all linked contents (scripts) are prefetched and mashed in order to perform deobfuscation. Therefore, the proposed system revolves around 3 modules: aggregation, deobfuscation, decision (analysis).

4.4 Recursive Pre-fetching of Suspicious Linked Contents

A malicious script is not expected to be monolithic and is often scattered across script files, DOM-embedded contents or iframes. In particular, linked contents injected via script or iframe tags may originate from different domains. Therefore, the script found in the original web page may not be malicious itself, or hard to be decided on. It is necessary to fetch these additional or linked contents in order to have an accurate view of the scripting contents involved in a web page. Another example is on-demand loading, where additional contents are only downloaded later during execution. Such behavior is implemented through the XmlHTTPRequest (XHR) object which allows asynchronous HTTP communication. Pre-fetching here will look for XHR and download script contents in advance to evaluate its impact on the current script contents. Prototype hijacking[32] has the potential to override a benign function with a malicious one.

Since malicious scripts may be obfuscated through several layers of obfuscations and redirections; the two first processing steps are recursive. Once a deobfuscation step has been taken, the outcome may be still obfuscated but the deciphering routine may not be directly present in the output but rather as a link, a DOM-embedded string or an iframe. Therefore, it is necessary to run the pre-fetching step again to gather scripting contents used in the deobfuscation step. These steps are repeated until the outcome is a plain interpretable script with no linked or DOM-embedded contents.

4.5 Emulation-based Deobfuscation

Obfuscation is any transformation that render a piece of code unreadable, hence hindering analysis by a human or an automated analyst. Obfuscating transformations span a wide set of techniques ranging from simple string splitting to encryption.

Obfuscated scripts, in particular, malicious obfuscated scripts carry out 4 common stages as stated in [11]: 1) redirection and cloaking, 2) deobfuscation, 3) environment preparation and 4) exploitation. This highlights the fact that some obfuscated scripts, e.g., encrypted ones, will eventually deobfuscate themselves before further execution. Such argument is always pointed out by dynamic analysis advocates to support the fact that obfuscation is not an issue, if not completely discarded by researchers that confuse obfuscation with malice. But the boundary between deobfuscation and the following stages is not always clear and some recently witnessed obfuscation schemes do interleave obfuscation with environment preparation and/or exploitation.

Contrary to past research that emulated a complete browser environment and just integrated a JavaScript engine, sometimes instrumented, we propose to emulate down to the execution of JavaScript, precisely at the deobfuscation stage (stage 2).

4.5.1 Approach

Since static methods cannot overcome the obstacle of obfuscation, emulation allows to reproduce the outcome of the deobfuscation stage without incurring side-effects inherent to a real execution. Here, the emulation is not aimed at the whole script and should only be carried out on the *obfuscated path*. The *obfuscated path* is comprised of all the instruc-

tions involved in deobfuscating the script: it can be further decomposed into the obfuscated strings and the deciphering routine (or decoder), when it exists. The first step of deobfuscation is therefore to extract the *obfuscated path* from the aggregated script parse tree, obtained after pre-fetching. But only the decoder is actually emulated, obfuscated strings being used as input.

In this research, we focus on JavaScript, which has been widely used in attack scenarios. JavaScript is both an object-oriented and functional language but the obfuscation schemes we are considering seldom make use of functional properties of JavaScript (variable promotion, variable substitution). Given that the deobfuscation stage cancels a prior obfuscation to provide an executable script to the next stage, the deobfuscation process is bound to *terminate*. We assume that the deobfuscation stage ultimately *converges* to a unique *normal* form: the unobfuscated original script.

Concerned with issues of performance, soudness and completeness, we considered several approaches to automate the deduction of obfuscated strings by the deciphering routine. Emulating JS instructions through another scripting language obviously suffers from lack of completeness as well as poor performance inherent to scripting languages in general. To satisfy properties of soundness and completeness, we considered formal approaches such as theorem proving, but state-of-the-art theorem provers can not completely emulate JS obfuscating transformations. Eventually, we turned to rewriting systems. Maude[10] is such rewriting framework whose underlying logic is *membership equational logic*. Meseguer[30] observed that equational logic is very well suited to give executable axiomatizations of imperative sequential languages. It was suggested to us that functional modules in Maude[10] could fit our requirements for sound and complete deduction of the outcome of JS instructions. As a matter of fact, computation in functional modules is accomplished using the equations as rewrite rules.

4.5.2 Maude and Membership Equational Logic

Functional modules satisfy the *membership equational logic* as well as the additional requirement of being confluent and terminating. Functional modules are thereby used to emulate deobfuscation: variables and objects are mapped to Maude's sorts, instructions are emulated through equations. Computation is realized by using these equations as rewrite rules applied to the obfuscated strings, until a canonical form is found, i.e., the deobfuscated script. The system supports the extraction and conversion steps to deobfuscate a script: at first, a preliminary analysis should allow the system to disambiguate obfuscated contents from the rest of the code, and then further isolate the deciphering routine and the obfuscated strings. The deciphering routine is then converted to a functional module's equations which are then used to rewrite the obfuscated string through reduction. Algorithm 1 gives a more detailed description of the processing that takes place just after pre-fetching. Once the employed obfuscation scheme has been detected, the *obfuscated path* is extracted and the deciphering routine is converted into a Maude functional module. Upon deobfuscation, an additional step verifies whether deobfuscation is still needed.

An advantage of Maude is that it employs term-indexing techniques to achieve high speeds of rewriting[37]. However, it does not support every JS native construct, such as loops. Yet, we can take advantage of conditional equations to emulate recursions. Doing so requires additional processing to transform JS loops to recursive functions. With a similar approach to some previous works in binary deobfuscation[1], we propose to apply automated deduction to the deobfuscation of scripting languages. In particular, for this work, we focus our efforts on JavaScript, a Web 2.0 de-facto standard language embedded natively in every browser and actively used in attack scripts. However, we believe that such approach can be applied to scripting languages bearing similar properties to JS, such as ActionScript (Flash) or VBScript.

Algorithm 1 Automated deduction of script instructions

```
1: obfstring, decroutine = extract(script)
2: if script contains loops then
3:     script = loopToRecursion(loops)
4: end if
5: fmod = convert(decroutine)
6: output = Maude.reduce(fmod,obfstring)
7: if output contains links then
8:     output += prefetch(links)
9: end if
10: if output is obfuscated then
11:     script = output
12:     repeat from line 1
13: end if
```

4.5.3 AST-based Extraction of Obfuscated Contents

Techniques to detect obfuscation in web scripts[9, 26] have been proposed recently. The goal is usu-

ally to detect malware assuming obfuscation is an indicator of malice. Both proposals make use of machine learning methods whether on statistical string features (byte occurence, entropy, word size) or semantic features (JS keywords and symbols).

Contrary to these related works, we are not considering the obfuscated string itself, but rather the obfuscating transformation or combination of obfuscating transformations, often implemented as automated tools. This proposal is not a method for deobfuscation however; it seeks to exhibit significant features of obfuscating transformations in order to automate their detection and deobfuscation.

In our approach, we consider the abstract syntax trees of scripts to analyze. We distinguish two distinct phases: first, we attempt to learn obfuscating transformations as subtrees in obfuscated scripts' ASTs; then, we try to identify the presence of such obfuscating transformations by matching learned patterns in candidate scripts' ASTs. The expression of a script as an AST is common to both phases and is obtained by simply parsing the script. Our subtree matching prototype developed in Ruby takes advantage of the *johnson* library [2] which supports the SpiderMonkey [13] JavaScript engine.

Since we are concerned with reducing the entropy of script contents for the purpose of analysis, it became necessary to abstract the script code in order to get rid of the randomization introduced in the identifiers and values. An accurate and abstract representation of a program is its abstract syntax tree. However, the AST used here is different in some aspects from similar approaches [12]:

- values are discarded and replaced by generic types: *NUM* for numeric values, *STR* for string values, *ID* for identifiers;
- some identifiers for core objects, e.g., `document`, and functions, e.g., `String.fromCharCode()`, are preserved in order to make explicit operations such as overriding or aliasing of core objects;
- conditions of branching and looping are discarded and the whole construct is replaced by a couple $\langle S, \mathcal{I} \rangle$ where S represents a symbol (either *BRANCH* or *LOOP*) and $\mathcal{I}$ the set of child nodes, which are instructions that form the body of the branch or loop;
- all instructions in a block are represented at the same level by sibling nodes, children of a node representing the containing block (which can be a branching control, a loop, a function definition, etc.).

In this approach, we focus on the hierarchical properties of tree-like structures.

Obfuscated contents represent part of the code of JS malware and can therefore be considered as subtrees of the malware's AST. Provided we can learn recurring subtrees from obfuscted JS samples, it is possible to match such patterns in other JS ASTs by subtree matching.

We can build a pushdown automaton (PDA) accepting the selected subtrees. Then, any analyzed script is transformed into an AST and fed to the PDA in order to match occurrences of the learned subtrees (the ones expressing obfuscation). This technique has been proposed by Flouri et al. [14], who aim to apply or adapt algorithms, commonly applied to strings and sequences, to the field of tree structures. Strings are usually processed using finite state machines as the model of computation and this naturally led to the use of pushdown automata towards trees, since the addition of a stack accommodates recursion. Thanks to syntactic rules that limit the number of combination between symbols in a programming language, employing a subtree automaton seems an effective solution that can scale with a large number of subtrees. Still, the size of these subtrees can badly affect performance and should be limited.

Constructing a deterministic pushdown automata accomodating a set of subtrees is done in three distinct steps:

1. construction of a PDA accepting a set of subtrees in their prefix notation
2. construction of a nondeterministic subtree matching PDA for a set of subtrees in their prefix notation
3. transformation of the nondeterministic subtree matching PDA to an equivalent deterministic PDA

Each step employs well-known PDA construction algorithms which are further detailed in [14]. In this research, we implemented a script that parses an XML file listing recurring subtrees and generates the transition rules for the deterministic subtree matching PDA. We also modified a program emulating a finite state machine to accommodate a stack. This program would parse the file containing the transition rules, as well as a file containing the script to be analyzed. The script to be analyzed is transformed into an AST and then expressed as its prefix notation to be fed to the subtree matching PDA. Whenever, there is a match, the matched subtree is reported, allowing to identify characteristic subtrees while traversing the script's AST.

```
<html>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /index.php was
not found on this server.</p>
<p>Additionally, a 404 Not Found
error was encountered while trying to use
an ErrorDocument to handle the request.</p>
<hr>
</body></html><script language=JavaScript>
str = "qndy`mh)(:" // the obfuscated string is
// abbreviated for the purpose of brevity
str2="";for (i = 0; i < str.length; i ++){
str2=str2+String.fromCharCode(str.charCodeAt(i)^1); };
eval(str2);</script></html>
```

Figure 4: Original HTML code

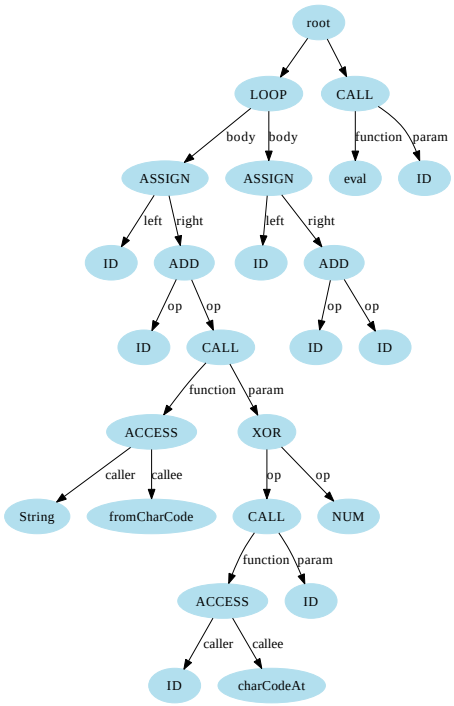

Figure 5: Abstract syntax tree of an eval unfolding transformation.

4.5.4 Example

This example (see Fig. 4) is a simple *eval unfolding* featuring a single loop that deciphers an obfuscated string via XOR operations. The script is included into an HTML file that displays a 404 error page to an unsuspecting user.

In the previous learning stage, we have learnt one instance of eval unfolding loop and its AST. Figure 5 displays a sample AST for the following eval unfolding loop:

```
for(i=0;i<str.length;i++){
  str2 = str2 +
  String.fromCharCode(str.charCodeAt(i)^1);
  str3 = str3 + str3};
eval (str2);
```

The LOOP statement contains two instructions represented by two paths stemming from the LOOP node: one being the actual decoder using the `String.fromCharCode` function and the other path being a dummy operation on an unrelated variable.

We first extract the script contents from the HTML file, i.e., instructions comprised between the `<script>` tags, and parses the contents. The parse tree is analyzed to detect the obfuscation scheme by PDA-based subtree matching. Here, the obfuscation scheme uses a loop to process the obfuscated string. This loop is converted to a recursive function whose body is the decoding routine. The Maude system readily provides a predefined functional module that defines the string data type as well as operators to manipulate string objects: the `fromCharCode()` function is mapped to Maude's `char` operator, which converts an ASCII code to the corresponding character; the `charCodeAt()` function is emulated by the combination of two basic operators, `ascii`, the inverse of `char`, and `substr`, the substring operator. The result of the conversion to a Maude functional module is displayed in

```
fmod TEST is
  protecting INT .
  protecting STRING .
  op test : Int String String -> String .
  var I : Int .
  vars S1 S2 : String .
  ceq test(I,S1,S2) = S2 if length(S1) <= I .
  ceq test(I,S1,S2) = test((I + 1),S1,S2)
  + char(ascii(substr(S1,(length(S1) - I - 1),1)) xor 1)
    if I < length(S1) .
endfm
```

Figure 6: Maude functional module

Figure 6. The workflow of the recursion is realized through conditional equations.

4.6 Decision

Static methods are difficult to apply to obfuscated strings. Our method here applies to a deobfuscated string obtained after emulation. The intuition is that we are able to tell what the script intends to do without executing it. By looking to what the script offers to do, i.e., its functionalities, it is possible to understand the actions a script might perform upon execution. To that end, script contents undergo static flow analysis on function calls or operators that are traced back to the root of an instruction block. This analysis allows building func-

tional units that decompose the script into blocks that express a single functionality. The combination or sequence of functionalities indicate what are the intentions of the script.

4.6.1 Static Functional Unit Decomposition

Algorithm 2 Block-based forward flow tracing

```
functions = []
params = []
transitions = []
for blk ∈ blocks do
  for instr ∈ instructions do
    if instr ⊇ call ∈ calls then
      functions[call].add(instr,blk)
      for par ∈ params do
        params[par].add(instr,blk)
      end for
    end if
  end for
end for
for f ∈ functions do
  for instr, blk ∈ f do
    for instrʹ, blkʹ ∈ fʹ do
      if (instr, blk) = (instrʹ, blkʹ) then
        transitions.add(instr,blk)
      end if
    end for
  end for
end for
transitions.sort()
```

A *functional unit* is a set of instructions that express a single functionality. The term was first coined by Lu and Kan[28] and originally refers to a JavaScript instance, combined with all of (potentially) called subprocedures. Algorithm 2 describes the steps taken to cluster instructions to functionalities and link these through their interaction. A block-based forward flow tracing approach is used to list native function calls in every block of the program as well as instructions related to these functions. Functions manipulating common variables are then seen as interacting, providing a logical link between two clusters.

While Lu and Kan favored a top-down approach, we advocate a bottom-up linking approach in order to focus on the functional characteristic of the clustered instructions. Besides, a JS function is not expected to perform a single functionality, especially malicious ones. Our method applies on the parse tree (or its abstract syntax tree (AST) representation) of a deobfuscated script and directly targets explicit functions. Explicit functions are functions of which functionality is obvious and that have been classified by us. By tracing the flow of these functions, we can identify clusters of instructions that express a single functionality (Fig. 7) and eventually trace the flow of variables through these different clusters. Such abstract model can be then compared with a database of models. At the time

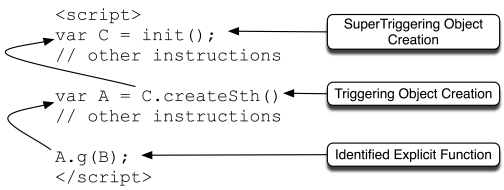

Figure 7: The proposed JavaScript functional unit

```
poexali();
function poexali() {
var ender = document.createElement('object');
ender.setAttribute('id','ender');
ender.setAttribute
('classid','clsid:BD96C556-65A3-11D0-983A-00C04FC29E36');
try {
var asq = ender.CreateObject('msxml2.XMLHTTP','');
var ass = ender.CreateObject("Shell.Application","");
var asst = ender.CreateObject('adodb.stream','');
try {
asst.type = 1;
asq.open('GET','http://attacksite//attack.php',false);
asq.send();
asst.open();
asst.Write(asq.responseBody);
var imya = './/../svchosts.exe';
asst.SaveToFile(imya,2);
asst.Close();
} catch(e) {}
try {
ass.shellexecute(imya);
} catch(e) {}
} catch(e) {}
}
```

download
storage
execution

Figure 8: Colored output after processing

of writing, we are not able to predict the average size of such models. Examples usually feature 3 or 4 functional units.

4.6.2 Example

The output of the Maude functional module (see Fig. 6) obtained in the previous step is then parsed and decomposed into functional units. This output is actually the result of emulating the decoding routine on the obfuscated string, which normally yield the original unobfuscated code. In this deobfuscated code, the `open()` call issued by the variable `asq`, which is an instance of the `msxml2.XMLHTTP` object allows clustering instructions dedicated to the download of some data, while the `SaveToFile()` call from the ADODB stream instance, `asst` isolates storage-related instructions. Finally, the `shellexecute()` function is linked to a `ShellApplication` instance, `ass` that expresses an execution functionality. The clustered instructions have been colored differently in a modified output (see Fig. 8) of the Maude framework.

This output can be further abstracted to a func-

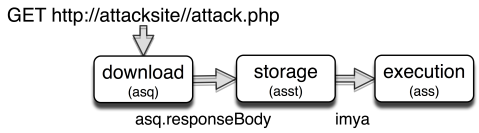

Figure 9: Functional model computed from the sample file

tional model as shown in Figure 9. This abstract view offers a more straightforward model of what activities the malicious script is actually carrying out: after downloading contents from a remote URL, the contents of the response, `asq.responseBody`, are passed to a storage functionality that saves these into a file called `imya` and this file is then inputted into an execution functionality.

5 Conclusion and Future Works

For its second year of activity, the SWAN WG is steadily progressing. So far, we have proposed some approaches to tackle Web 2.0 security issues with our JS analysis proxy, as well as a federating project that is the Web2Sec testbed.

In particular, the proxy is entering an advanced stage in its design since all modules are completely defined. So far, its implementation has yielded promising proof-of-concepts. However, this still needs a little more work to reach completion, and obviously further evaluation is needed. Numerical results and discussion can be found in the produced publications[4, 3, 5].

As for the Testbed, it is in stand-by mode but should be resumed in a near future.

We are still calling for other participants to propose new projects or contribute on exisiting ones. In particular, we are interested in developing a JS debugger, as well as other analysis tools targeted to other aspects of web applications (Flash, PDF, etc.) and web services. New approaches to analysis such as formal methods or symbolic execution also offer plenty of research opportunities and perspectives.

References

[1] R. Ando. Parallel Analysis of Polymorphic Viral Code Using Automated Deduction System. In *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2007.

[2] J. Barnette. johnson. Available at: `http://github.com/jbarnette/johnson/`.

[3] G. Blanc, M. Akiyama, D. Miyamoto, and Y. Kadobayashi. Identifying Characteristic Syntactic Structures in Obfuscated Scripts by Subtree Matching. In *Proceedings of the anti Malware engineering WorkShop (MWS 2011)*, Oct. 2011.

[4] G. Blanc, R. Ando, and Y. Kadobayashi. Term-Rewriting Deobfuscation for Static Client-Side Scripting Malware Detection. In *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011)*, Feb. 2011.

[5] G. Blanc and Y. Kadobayashi. A Step Towards Static Script Malware Abstraction: Rewriting Obfuscated Script with Maude. *IEICE Transactions on Information and Systems*, E94-D(11):2159–2166, Nov. 2011.

[6] Bonsai Information Security. moth. `http://www.bonsai-sec.com/en/research/moth.php`.

[7] E. Bursztein et al. Webseclab Security Education Workbench. In *Proceedings of the 3rd Workshop on Cyber Security Experimentation and Test*, Aug. 2010.

[8] R. Cannings, H. Dwidedi, and Z. Lackey. *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions*. McGraw-Hill, 2007.

[9] Y. Choi, T. Kim, S. Choi, and C. Lee. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *Proceedings of the 1st International Conference on Future Generation Information Technology*, pages 160–172. Springer-Verlag, 2009.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí Oliet, J. Meseguer, and C. Talcott. The Maude System. Available at: `http://maude.cs.uiuc.edu`.

[11] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by Download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International WWW Conference*, 2010.

[12] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[13] B. Eich. SpiderMonkey (JavaScript-C) Engine. Available at: `http://www.mozilla.org/js/spidermonkey/`.

[14] T. Flouri, J. Janoušek, and B. Melichar. Subtree Matching by Pushdown Automata. *Computer Science and Information Systems*, 7(2):331–357, Apr. 2010.

[15] J. J. Garrett. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, 2005.

[16] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, May 2008.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International World Web Wide Conference (WWW 2009)*, Apr. 2009.

[18] B. Hoffman and B. Sullivan. *Ajax Security*. Addison-Wesley (Pearson), 2007.

[19] A. Holdoner. *Ajax: The Definitive Guide*. O'Reilly, 2008.

[20] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *Proceedings of 18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, Mar. 2004.

[21] C. Jackson and H. J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proceedings of the 16th International World Web Wide Conference (WWW 2007)*, May 2007.

[22] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-Enforced Embedded Policies. In *Proceedings of the 16th International World Web Wide Conference (WWW 2007)*, May 2007.

[23] M. Johns. On JavaScript Malware and Related Threats. *Journal in Computer Virology*, 4(3):161–178, Aug. 2008.

[24] M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In *Proceedings of the OWASP Europe 2006 Conference*, May 2006.

[25] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers. In *Proceedings of the 17th International World Web Wide Conference (WWW 2008)*, Apr. 2008.

[26] P. Likarish, E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *4th International Conference on Malicious and Unwanted Software*, pages 47 – 54, Oct. 2009.

[27] G. Louthan, W. Roberts, M. Butler, and J. Hale. The Blunderdome: An Offensive Exercise for Building Network, Systems, and Web Security Awareness. In *Proceedings of the 3rd Workshop on Cyber Security Experimentation and Test*, Aug. 2010.

[28] W. Lu and M.-Y. Kan. Supervised Categorization of JavaScript using Program Analysis Features. In *Asian Information Retrieval Symposium*. Springer-Verlag, 2005.

[29] S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, 2009.

[30] J. Meseguer. Software Specification and Verification in Rewriting Logic. *Models, Algebras and Logic of Engineering Software*, 2003.

[31] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of the 16th Annual USENIX Security Symposium*, 2007.

[32] S. D. Paola and G. Fedon. Subverting Ajax for Fun and Profit. In *Proceedings of the 23rd Chaos Communication Congress*, 2006.

[33] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*, Apr. 2007.

[34] M. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in Circumventing Web-Malware Detection. Technical Report rajab-2011a, Google, Inc,, July 2011.

[35] C. Reis, J. Dunagan, H. J. Wang, and O. Dubrovsky. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[36] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[37] N. Shankar. Automated Deduction for Verification. *ACM Computing Surveys*, 41(4), Oct. 2009.

[38] P. Vogt et al. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium (NDSS 2007)*, Feb. 2007.

[39] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, Jan. 2007.