第 XII 部

ウェブアプリケーションの
セキュリティ技術の研究

第12部 ウェブアプリケーションのセキュリティ技術の研究

第1章 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 projects 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)[63], were technologies designed in the mid-1990s. 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 [81].

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[90]. More information on Web 2.0 vulnerabilities and related attacks can be found in [31, 80].

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 and 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. Thereby, recent research works have been carried out on the client-side exclusively, except for secure mashup schemes[87, 98].

Some researchers have concentrated on the browser itself where many vulnerabilities lie and where exploitation takes place: initiatives aiming to sandbox the browser[64] or to enforce security policies in the browser[89] have been proposed. Other works focused on how to prevent some kind of attacks such as XSS[86] or CSRF[91] by designing some heuristics to characterize these attacks. To a much more granular level, JavaScript being the de-facto standard in AJAX and other programming frameworks, attackers have naturally took advantage of it being enable 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[110], interposition[149, 195] 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[133, 145], data tainting to prevent XSS[184] or even control flow analysis[65].

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

第 2 章　Initial Year Activities

For 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. The first official meeting took place during the WIDE Project Spring Camp 2010 where we introduced Web 2.0 application security issues to WIDE members and launch the beginning of our 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.

Among the different approaches stated in Section 1.3, we were particularly active in analyzing JavaScript malware. For that purpose, we designed and started developing a proxy-based solution to analyze JS malware. This led to two publications in JWIS 2010 (August)[19] and AINTEC 2010 (November)[20], that will be detailed later.

第 3 章　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

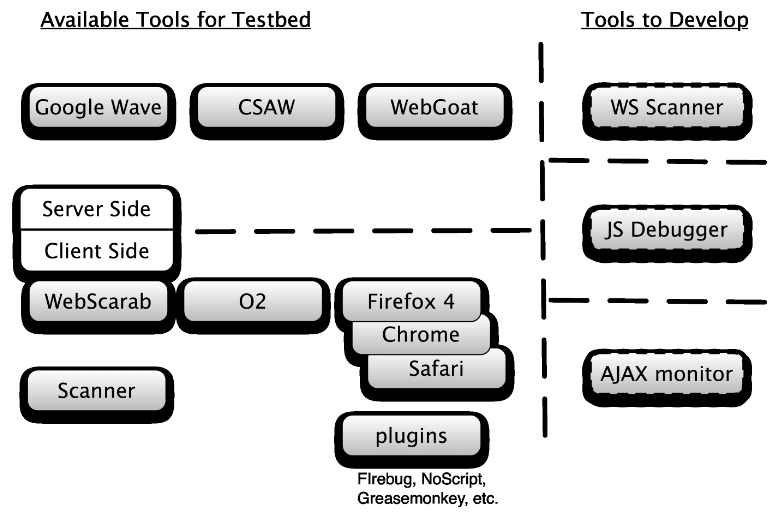

Fig. 3.1. Overview of the tools to be deployed in the Web2Sec testbed

deploy a testbed to perform experiments on Web 2.0 security issues. Similar to previous works from Stanford University[26], 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 3.1 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 3.1, 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 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[26] 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[108] which is rather an academic multi-layer service offensive security testbed simulating a university network and moth[23], a VM image that contains a set of vulnerable web applications and scripts for testing web application security scanners or static code analysis tools.

第 4 章　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 by injecting remote JS loading payload in 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.

We propose an approach featuring abstract interpretation to perform JS program static analysis on deobfuscated programs. We also propose a 3-module system to process JS-malware-infected web page contents in realtime to extract the deobfuscated payload and analyze it.

4.2 Threat Model

A common scenario unfolds as follows (cf. Fig. 4.1):

- 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

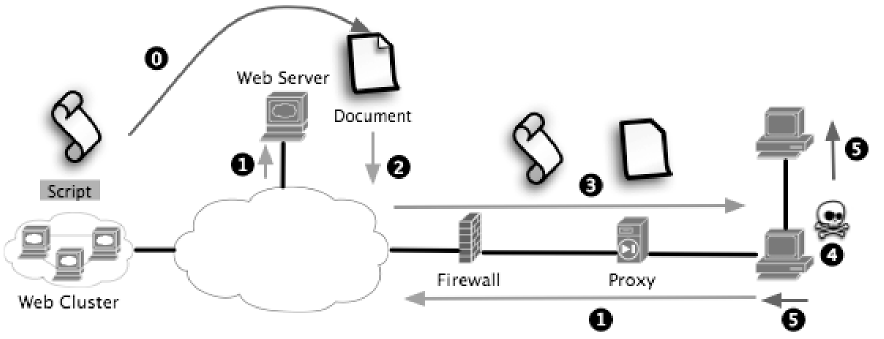

Fig. 4.1. Typical Web infection threat model scenario

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[145], we need to 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 which would sustain the load of realtime static analysis on JS candidates. The system is designed to work as follows (please refer to Fig. 4.2): 1) a victim is tricked in accessing a malicious link and subsequently request a web page containing malicious JavaScript (or not); 2) the requested page is sent to the user and intercepted by the proxy which will carry out the first step: page contents aggregation; 3) upon fetching

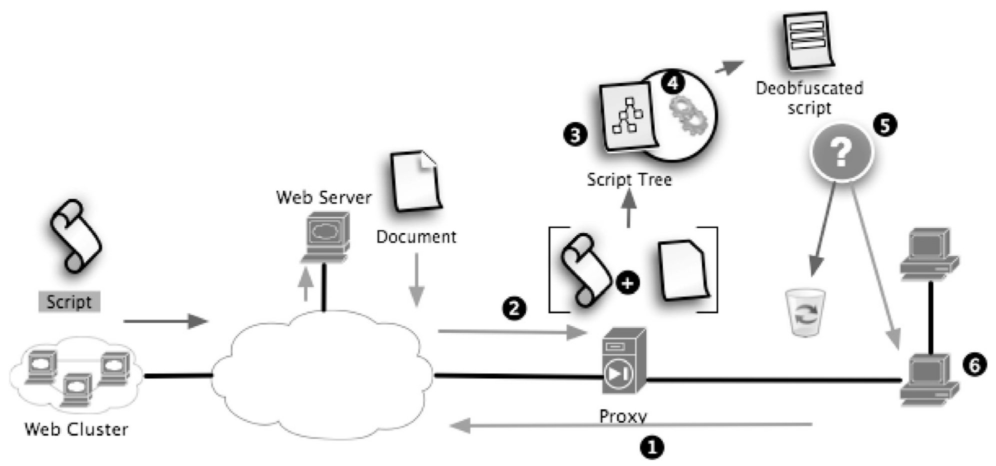

Fig. 4.2. Overview of the Proposed Solution

and aggregating contents, JavaScript contents are extracted and the rest of the HTML will serve as a context (as well as browser information). Extracted JavaScript is then computed as a tree representation; 4) using analysis techniques, the tree is flattened during the deobfuscation stage. Flattening the tree is assumed to be equivalent to "resolving" the deciphering routine; 5) the deobfuscated script is then provided to a decision algorithm; 6) upon decision, the whole aggregated contents might either be thrown away if it was detected as malicious (and then the user gets notice of such event) or forwarded to the user's browser, saving download time; 6bis) when on-demand JavaScript gets downloaded, script running on the user's browser is analyzed once more, with the addition of the new or updated script contents.

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) is prefetched and mashed in order to perform deobfuscation. Therefore, the proposed system revolves around 3 modules: aggregation, deobfuscation, decision (analysis).

4.4 Deobfuscation

According to [12], the goal of program obfuscation is to make a program unintelligible while preserving its functionality. Based on that definition, it is pretty obvious why such techniques have been widely used in recent JavaScript-based attacks. It is both a way for attackers to preserve the intellectual property of their payloads and a way for exploit kit users to prevent analyzers from learning noticeable patterns. A good survey on past and actual web malware obfuscation techniques can be found in [38].

We propose to tackle the obfuscation issue in order to provide a deobfuscated program that will be further analyzed to determine whether or not it is malicious, reducing the number of false positives compared to previous script analysis approaches. We also envision to provide a more sound and complete analysis by relying on a static approach to perform deobfuscation. According to [38], obfuscation is somehow reversible. Since the code needs to be executed on the client-side, in no way, an encryption scheme would be applied without providing the decryption key at some point or another. Therefore, there is a possibility to retrieve this key and to carry out deobfuscation. However, this key might not be immediately available, and it is a property of modern obfuscation schemes that decoding might occur after several layers of (de)obfuscation or redirection.

4.4.1 Approach

Though dynamic analysis methods involving execution-based deobfuscators and JavaScript engines have been popular, they are lacking in code coverage. In particular, obfuscation techniques are more and more sophisticated and employ several anti-analysis tricks to evade analysis. Good examples are context-dependent encryption schemes. Such context is natively present in the browser but it is not safe to execute possibly malicious contents in the browser without providing a robust sandbox. Indeed, some attack codes do not rely on classic critical sinks or do not use ciphering schemes at all and rather combine simple techniques to interleave deobfuscation and exploitation stages. A good tradeoff is therefore to delegate analysis to a proxy alleviating processing overhead and execution risk on the browser. A proxy might also accommodate several end hosts with different browser personalities, which it might not be able to impersonate. This affects the capacity of the proxy to replicate the context of execution. By using a static approach, we can partly provide the context (from the server side) without harming the code coverage of the analysis.

In this research, we envision to provide an iterative and static deobfuscation scheme that relies on the deduction of the result of a set of selected instructions. In a previous stage, we will also provide prefetching of linked contents to ensure that any remotely located library or script used for decoding will also be taken into account in our deobfuscation stage.

This approach is intutitive in that we are attempting to rewrite instructions to simplify the expression of a program to its normal, which we assume to be the deobfuscated form. This approach is easy to understand in the case of simple obfuscation techniques (such as string splitting for example) where we can recover the original deobfuscated code without the risk of executing it.

4.4.2 Term Rewriting

With a similar approach to some previous works in binary deobfuscation[5], 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.

In [5], binary instructions were transformed into first-order logic clauses and then passed as the input set of support to the Otter[124] theorem prover. The goal is to detect deciphering instructions by matching the resolved clause with a passive list of conclusion clauses. This proposal can detect instructions obfuscated by register substitution, provided that a passive list of goals, as well as paramodulation clauses, are learned beforehand.

Rewriting systems are powerful tools that can perform well, provided that some requirements are fulfilled:

- places to rewrite: it is important to define a strategy in order to perform timely. Indeed, it is costly to rewrite everything so it is necessary to focus only on what is obfuscated;
- rewrite rules: a theorem prover is only able to manipulate first-order logic clauses. Moreover, it also needs to be told how to rewrite a set of clauses by specifying rules. These rules would support the system in handling function natives to JS, that are not understood by the theorem prover;
- termination property: a rewriting system simplifies terms to a form where they cannot

be further rewritten, known as *normal form*. This is linked to the property of *confluence* where terms can be rewritten by different rules in different forms that ultimately yield to the same result. A *noetherian* rewriting system terminates providing a normal form, which we assume to be the deobfuscated form.

4.4.3 Overview

Our proposed system is a rewriting system as described in Figure 4.3. The rewriting system takes as input the obfuscated code and the decoding code, as well as the rewrite rules. A preprocessing stage traces parameters from critical sinks, if available, in order to select instructions to be rewritten. Clauses are generated from transforming scripts instructions. Such instructions can be clustered if they manipulate independent parameters, allowing process parallelization. Clauses fed to the theorem prover are deduced to simpler clauses by applying rewrite rules but some transformation such as string splitting or variable substitution can be resolved using *demodulation*.

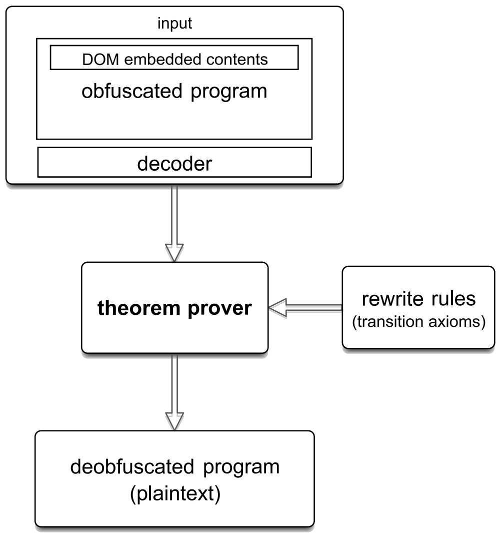

Fig. 4.3. Overview of the rewriting system

4.4.4 Example

As evidence to our assumptions, we display here an attempt to partially apply automated deduction to rewrite a JS program. A snippet of the

```
document.write('<p>1003</p>');
var hoglev=
    parseInt(document.getElementById('ityper')
    .getElementsByTagName(String
    .fromCharCode(0x70))[parseInt(String
    .fromCharCode(0x30))].innerHTML);
var huawcu='';
for (ipjrbt = hoglev; ipjrbt > 0; ipjrbt--)
{
    for (cvjla = hoglev-ipjrbt;
        cvjla <= yoylk.length;
        cvjla=cvjla+hoglev)
    {
        huawcu=huawcu+yoylk.charAt(cvjla);
    }
}
var ewhne4=huawcu+"SMB();";
var eoyqyw=csblj6(ewhne4);
eval(eoyqyw);
```

Fig. 4.4. An obfuscated JavaScript code

```
given clause #8: (wt=15) 10 [] -dom_operation_1(x,y)|
-dom_operation_2(w,z)| -object(x2)|
dom_operation(x,object(x2),y,w,z).
** KEPT (pick-wt=8): 15 [hyper,10,13,14,12]
dom_operation(getElementsByTagName,object(BODY),
p,parseInt(0),innerHTML).

----> UNIT CONFLICT at   0.00 sec ---->
16 [binary,15.1,11.1] $F.


-------- PROOF --------
Length of proof is 4.  Level of proof is 2.

---------------- PROOF ----------------

1 [] eq(fromCharCode(ascii(70)),p).
2 [] eq(fromCharCode(ascii(30)),0).
3 [] getElementById(ityper)=object(BODY).
4 [] dom_operation_2(getElementsByTagName,x,y)=
dom_operation_2(x,y).
5 [] String(x)=x.
6 [] parseInt(document(object(BODY)))=
object(BODY).
7 [] parseInt(document(getElementById(ityper))).
8 [] dom_operation_1(getElementsByTagName,
String(fromCharCode(ascii(70)))).
9 [] dom_operation_2(getElementsByTagName,
parseInt(String(fromCharCode(ascii(30)))),innerHTML).
10 [] -dom_operation_1(x,y)| -dom_operation_2(w,z)|
-object(x2)|dom_operation(x,object(x2),y,w,z).
11 [] -dom_operation(x1,x2,x3,x4,x5).
12 [7,demod,3,6] object(BODY).
13 [8,demod,1,5] dom_operation_1(getElementsByTagName,
p).
14 [9,demod,2,5,4] dom_operation_2(parseInt(0),
innerHTML).
15 [hyper,10,13,14,12]
dom_operation(getElementsByTagName,object(BODY),p,
parseInt(0),innerHTML).
16 [binary,15.1,11.1] $F.

------------ end of proof -------------


Search stopped by max_proofs option.
```

Fig. 4.5. Output of the resolution by Otter of a set of JavaScript instructions

source code we used for this sample analysis is shown in Figure 4.4.

We manually transformed instructions to clauses and passed these clauses to the Otter theorem prover in order to perform demodulation (a restricted form of paramodulation). We present the output in Figure 4.5.

For this proof-of-concept, we did set a `max_proofs` option in order to halt the search after a certain number of rounds. However, since the deobfuscation is bound to terminate (as it outputs the deobfuscated code), it may stops after a fewer or a longer number of rounds. Here the output clearly indicates that, so far, this example resolution scales to the millisecond. Obviously, we still need to work around some issues.

```
poexali();
function poexali() {
var ender = document.createElement('object');
ender.setAttribure('id','ender');
ender.setAttribure('classid','c'+'l'+"sid:B"+
 "D9"+'6C556-65'+"A3-11"+'D0-983A-0'+"0C"+
 '04F'+"C29"+'E36');
try {
var asq = ender.CreateObject('m'+"sx"+'ml2'+
 "."+'X'+"ML"+'H'+'TTP','');
var ass = ender.CreateObject("Sh"+"ell.A"+"p"+
 "plica"+"tion",'');
var asst = ender.CreateObject('a'+'d'+"odb."+
 'st'+"r"+'eam','');
try {
asst.type = 1;
asq.open('G'+"E"+'T','<URL>',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){}}
```

Fig. 4.6. Drive-by Download Attack Payload

4.5 Decision

We have the intuition that JS programs can be decomposed in a sequence of basic program functionalities and that some combinations of functionalities can indicate malice. Past research works[109] indicate JS programs can be decomposed as *functional units* or simply, units. These units are defined as JavaScript instances, combined with all their (potentially) called subprocedures. Interestingly, such construct relies on the flow of calls initiated by the event handler. Our scope is different in that we concentrate on the flow of instantiated objects and their interaction to define functional units.

4.5.1 Approach

Constructing such model can be performed through the extraction of an **abstract semantic graph** or **ASG**. The ASG would then be further abstracted by clustering operations that manipulate the same *objects*. For example, the sample in Figure 4.6 can be represented as an ASG. Each branch and each instruction are therefore browsable. We can cluster instructions that manipulate the same parameters or variables and reduce the ASG to functional units. Instructions pertaining to one or several clusters are inferred as logical interfaces between these clusters. We can then clearly see the sequence and combination of functionalities leading to a drive-by download attack. Another application is the comparison with known JS malware models. Previous research works indicate the relevance of such technique in the field of code difference analysis[147].

4.5.2 Functional unit

A functional unit is a set of instructions that expresses the same functional activity (e.g., download, storage, execution, etc.). To achieve such abstraction, we need a semantic understanding of the analyzed scripting language. Therfore, we need to define a semantic for the targeted scripting language in order to classify native objects, functions to a finite set of functions.

4.5.3 Overview

Based on previous works from Cova et al.[37] that stipulate that Web-based malware are performing through 4 distinct stages (namely, 1) redirection and cloaking, 2) obfuscation, 3) environment preparation, 4) exploitation), our decision module concentrates on the last two stages to analyse and decide whether a JS program is malicious. In particular, we have proposed to

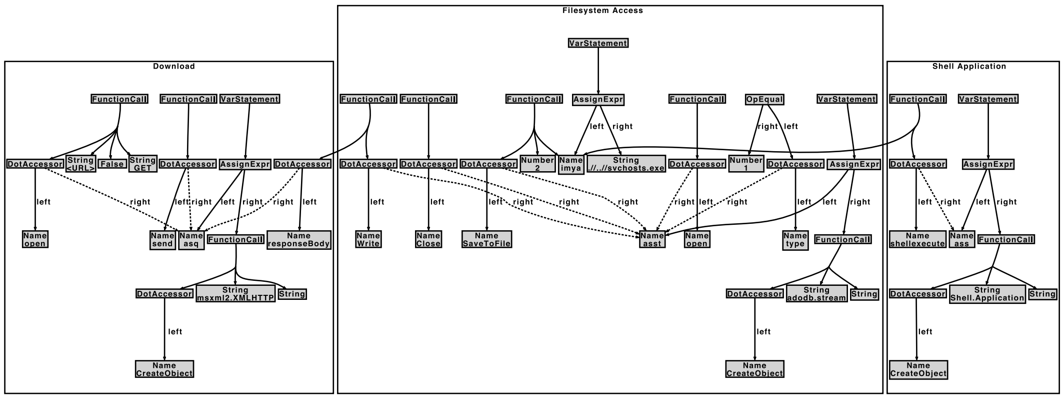

Fig. 4.7. Abstract Semantic Graph of a Drive-by Download Attack

abstract the deobfuscated program as a sequence of functional units. The idea is to conduct static data-flow analysis to extract functional units as well as the interfaces between these different units. Using semantics specific to JavaScript functions and API functions we wish to clusterize instructions based **on what they actually do** and on which data they are manipulating. The result can be displayed as in Figure 4.6 or as an ASG for comparison purposes. In Figure 4.7, we display an example of an ASG built from the program displayed in Figure 4.6. We can clearly see the relation between the different object instances and how data flows from one object to another. By following this flow and clusterizing instructions around object instances, we can clearly decompose this program into 3 functional units. In Figure 4.7, 3 objects are instantiated: `asq` which is an `msxml2.XMLHTTP` object; `asst` which is an `adodb.stream` object; `ass` which is a `Shell.Application` object. Based on semantics specific to JavaScript APIs, we know that an XMLHTTP object carries out HTTP communication and is then used to download contents from remote web servers, while the ADO stream object is used to manage a stream of binary data and the Application property of the Shell automation object allows the execution of an instance of an application. In particular, here the stream object uses the Write function, thereby we can further characterize these instructions as a data stream storage unit. Eventually, by following the flow of data, our proposed system can deduce that the program carries out the following functionalities: it downloads data then stores this data and finally executes the stored data.

第 5 章　Conclusion and Future Works

For its first year of activity, the SWAN WG is slowly but surely starting projects. 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. We already began to acquire visibility but surely needs to improve on some aspects of our proposal: the proxy deobfuscation module needs more thorough evaluation and the decision module is still to implemented. The Web2Sec testbed has not yet been into motion and that is one bad point so far. However, we will try as much as possible to stick to our schedule as show in Figure 5.1. Hopefully, this will triggers more publications and we hope to actively enter our second year of activity.

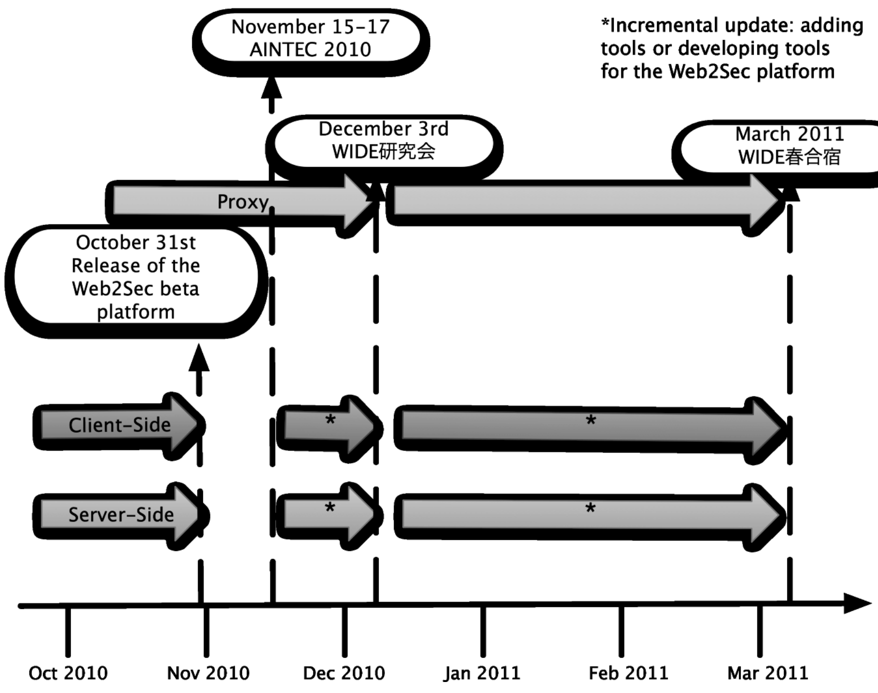

Fig. 5.1. SWAN WG provisional schedule