Semantics-Preserving Dissection of JavaScript Exploits via Dynamic JS-Binary Analysis

Xunchao Hu[1], Aravind Prakash[2], Jinghan Wang[1], Rundong Zhou[1], Yao Cheng[1], and Heng Yin[1](✉)

[1] Department of EECS, Syracuse University, Syracuse, USA
{xhu31,jwang153,rzhou02,ycheng,heyin}@syr.edu

[2] Computer Science Department, Binghamton University, Binghamton, USA
aravind@cs.binghamton.edu

Abstract. JavaScript exploits impose a severe threat to computer security. Once a zero-day exploit is captured, it is critical to quickly pinpoint the JavaScript statements that uniquely characterize the exploit and the payload location in the exploit. However, the current diagnosis techniques are inadequate because they approach the problem either from a JavaScript perspective and fail to account for "implicit" data flow invisible at JavaScript level, or from a binary execution perspective and fail to present the JavaScript level view of exploit. In this paper, we propose JScalpel, a framework to automatically bridge the semantic gap between the JavaScript level and binary level for dynamic JS-binary analysis. With this new technique, JScalpel can automatically pinpoint exploitation or payload injection component of JavaScript exploits and generate minimized exploit code and a Proof-of-Vulnerability (PoV). Using JScalpel, we analyze 15 JavaScript exploits, 9 memory corruption exploits from Metasploit, 4 exploits from 3 different exploit kits and 2 wild exploits and successfully recover the payload and a minimized exploit for each of the exploits.

Keywords: Exploit analysis · Malicious JavaScript

1 Introduction

Malicious JavaScript has become an important attack vector for software exploitation attacks. Attacks in browsers, as well as JavaScript embedded within malicious PDFs and Flash documents, are common examples of how attackers launch attacks using JavaScript. Interactive nature of JavaScript allows malicious JavaScript to take advantage of binary vulnerabilities (e.g., use-after-free, heap/buffer overflow) that are otherwise difficult to exploit. In 2014, 639 browser vulnerabilities were discovered and the number was increased by 8 % over 2013 reported by Symantec [5]. This provides the attacker a broad attack space.

Previously unknown, or "zero-day", exploits are of particular interest to the security community. Once a malicious JavaScript attack is captured, it must be analyzed and its inner-workings understood quickly so that proper defenses

F. Monrose et al. (Eds.): RAID 2016, LNCS 9854, pp. 254–276, 2016.
DOI: 10.1007/978-3-319-45719-2_12

can be deployed to protect against it or similar attacks in the future. Unfortunately, this analysis process is tedious, painstaking, and time-consuming. From the analysis perspective, an analyst seeks to answer two key questions: (1) Which JavaScript statements uniquely characterize the exploit? and (2) Where is the payload located within the exploit? The answer to the first question results in the generation of an exploit signature, which can then be deployed via an intrusion detection system (IDS) to discover and prevent the exploit. The answer to the second question allows an analyst to replace the malicious payload with an amicable payload and use the modified exploit as a proof-of-vulnerability (PoV) to perform penetration testing.

Program slicing [34] is a key technique in exploit analysis. This technique begins with a source location of interest, known as slicing source, such as a statement or instruction that causes a crash, and identifies any statements or instructions that this source location depends on. Prior exploit analysis solutions have attempted to analyze exploits at either the JavaScript level [11,12,18,20,26,27] or the underlying binary level [23,24,31,36,38].

While binary level solutions execute an exploit and analyze the underlying binary execution for anomalies, they are unaware of any JavaScript level semantics and fail to present the JavaScript level view of the exploit. JavaScript level analysis fails to account for implicit data flows between statements because any DOM/BOM APIs invoked at the binary level are invisible at the JavaScript level. Unfortunately, implicit flows are quite common in attacks and are often comprised of seemingly random and irregular operations in the JavaScript that achieve a precise precondition or a specific trigger which exploits a vulnerability in the binary. The semantic gap between JavaScript level and binary level during the analysis makes it challenging to automatically answer the 2 key questions.

In this paper, we present JScalpel with password: "artifacts", a system that creatively combines JavaScript and binary level analyses to analyze exploits. It stems from the observation that seemingly complex and irregular JavaScript statements in an exploit often exhibit strong data dependencies in the binary. JScalpel utilizes the JavaScript context information from the JavaScript level to perform *context-aware* binary analysis. Further, it leverages binary analysis to account for implicit JavaScript level dependencies arising due to side effects at the binary level. In essence, it performs JavaScript and binary, or *JS-Binary* analysis. Given a functional JavaScript exploit, JScalpel performs JS-Binary analysis to: (1) generate a minimized exploit script, which in turn helps to generate a signature for the exploit, and (2) precisely locate the payload within the exploit. It replaces the malicious payload with a friendly payload and generates a PoV for the exploit.

We evaluated JScalpel on a corpus of 15 exploits, 9 from Metasploit[1], 4 exploits from 3 different exploit kits and 2 wild exploits. On average, we were able to reduce the number of unique JavaScript statements by 49.8 %, and precisely identify the payload, in a semantics-preserving manner, meaning that

[1] Metasploit Framework – http://www.metasploit.com/, a popular penetration testing framework.

the minimized exploits are still functional. In addition, we were able to replace the payload with amicable payload to perform penetration testing. JScalpel showed an average reduction of 75.5 % in trace size and 16x improvement in time taken to trace. Finally, we presented the wild exploit CVE-2011-1255 as a case study. We demonstrate how the exploit is minimized and payload is located.

Contributions. We make the following contributions:

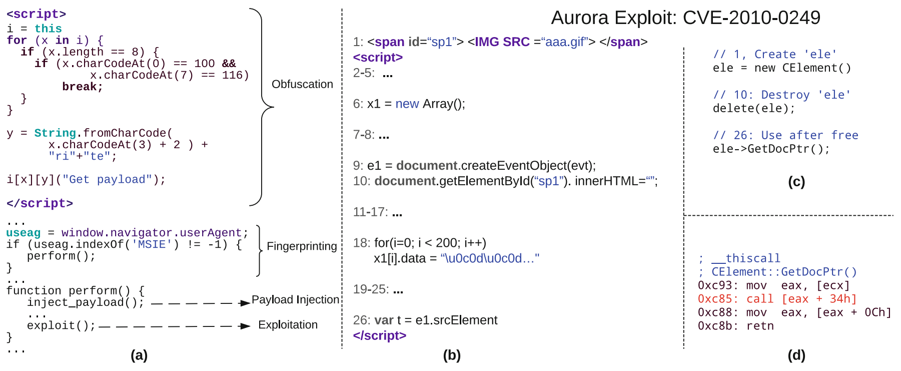
- We make a key observation that semantics-preserving JavaScript exploit analysis must bridge the gap between JavaScript and binary level.
- We propose a technique to bridge the semantic gap and tackle several challenges (e.g. dependency explosion and script engine code filtering) and incorporate our techniques into the JScalpel analysis framework.
- Using JScalpel, we analyze 15 JavaScript exploits, 9 memory corruption exploits from Metasploit, 4 exploits from 3 different exploit kits and two exploits from the wild and successfully recover the payload and a minimized exploit for each of the exploits.

2 Background and Overview

2.1 Components of JavaScript Attack

Modern JavaScript attacks can be divided into four general components. Figures 1(a) and (b) show these four components within the Aurora exploit.

Obfuscation: To avoid detection, obfuscation techniques are widely deployed in JavaScript attacks. For example, in Fig. 1(a) JavaScript obfuscation is used to perform a `document.write(‘‘Get payload’’)` operation. Simple static analysis-based scanners cannot identify that “i[x][y]” is actually a `document.write()` operation.



```
<script>
i = this
for (x in i) {
  if (x.length == 8) {
    if (x.charCodeAt(0) == 100 &&
            x.charCodeAt(7) == 116)
        break;
  }
}

y = String.fromCharCode(
      x.charCodeAt(3) + 2 ) +
      "ri"+"te";

i[x][y]("Get payload");

</script>
...
useag = window.navigator.userAgent;
if (useag.indexOf('MSIE') != -1) {
   perform();
}
...
function perform() {
   inject_payload();
   ...
   exploit();
}
...
```

```
1: <span id="sp1"> <IMG SRC ="aaa.gif"> </span>
<script>
2-5:  ...

6: x1 = new Array();

7-8: ...

9: e1 = document.createEventObject(evt);
10: document.getElementById("sp1"). innerHTML="";

11-17: ...

18: for(i=0; i < 200; i++)
     x1[i].data = "\u0c0d\u0c0d..."

19-25: ...

26: var t = e1.srcElement
</script>
```

```
// 1, Create 'ele'
ele = new CElement()

// 10: Destroy 'ele'
delete(ele);

// 26: Use after free
ele->GetDocPtr();
```

```
; __thiscall
; CElement::GetDocPtr()
0xc93: mov  eax, [ecx]
0xc85: call [eax + 34h]
0xc88: mov  eax, [eax + 0Ch]
0xc8b: retn
```

Fig. 1. (a) describes the components of a modern exploit, (b) presents the relevant JavaScript code involved in Aurora Exploit and (c) presents the underlying code execution that results in use-after-free, (d) presents the assembly code for function GetDocPtr.

Fingerprinting: An exploit uses fingerprinting to glean information about victim's environment. With such information, exploits specific to vulnerable components are launched to compromise the victim process. For example, in Fig. 1(a), the Aurora exploit is only performed if the type of the browser is identified as being Microsoft Internet Explorer ("MSIE").

Payload Injection: The exploit injects a malicious payload into the victim process. Payloads can be broadly categorized as executable or non-executable payloads. presents the payloads and the flow of execution in modern exploits. The goal of exploits is to execute a malicious payload, but since the wide deployment of data execution prevention (DEP), the page containing the executable payload cannot be directly executed. First, return-oriented programming (ROP) is used to make a page executable by invoking `VirtualProtect()` on Windows or `mprotect()` on Linux. Then, control is transferred to the malicious executable code.

Exploitation: In this step, using one or more carefully crafted JavaScript statements, the vulnerability in the victim process is exploited. The statements may seem random and may lack data-dependencies, but they often involve a combination of explicit and implicit data dependency. Consider the exploit statements for the Aurora exploit presented in Fig. 1(b, c and d). (b) presents the HTML (statement 1) and JavaScript (2–26) statements that exploit a use-after-free vulnerability in `mshtml.dll` of Internet Explorer browser. Figures 1(c and d) present the underlying C++ and assembly code that is executed as a part of the exploit. Statement 18 corrupts the memory that was freed in statement 10. The corrupted memory is utilized in a `call` instruction arising from statement 26. All the statements in Fig. 1(c) are pertinent to the exploit.

2.2 Problem Statement

We aim to develop JSCALPEL– a framework to combine JavaScript and binary analyses to aid in analysis of JavaScript-launched memory corruption exploits. It is motivated by two key observations.

First, analysis performed at only the JavaScript level is insufficient. In Fig. 1(b), JavaScript level analysis of Aurora captures the explicit data dependencies between statements 9 and 26 and statements 6 and 18. However, because no explicit dependency exists between statements 18 and 26, the two groups of statements will be incorrectly deemed to be independent of each other. Second, while complete, analysis performed at only the binary level is also insufficient. In Fig. 1(d), binary level analysis can expose the manipulation of pointers, however it can not expose exploit-related JavaScript statements in Fig. 1(c) due to the lack of JavaScript context. A binary-level analysis will show the memory written by the binary instructions of statement 18 is utilized through reads performed by binary instructions of statement 26, revealing a straight-forward data dependency between statements 18 and 26.

Input: JScalpel accepts a raw functional exploit and a vulnerable program as input. The vulnerable program can be any program like (PDF reader, web browser, etc.) as long as it can be exploited through JavaScript. The exploit consists of HTML and malicious JavaScript components. The exploit can be obfuscated or encrypted. JScalpel makes no assumptions about the nature of payloads. That is, the payload could be ROP-only, executable-only or combined.

Output: JScalpel performs JS-Binary tracing and slicing and generates 3 specific outputs. (1) A simplified exploit HTML that contains the key JavaScript statements that are required to accomplish the exploit, and (2) the precise JavaScript statements that inject the payload into the vulnerable process' memory along with the exact payload string – both non-executable and executable – within the JavaScript. Finally, (3) an HTML page, where the malicious payload is replaced by a benign payload is generated as a Proof-of-Vulnerability (PoV).

Delta debugging [37] is firstly proposed to generate the minimized *C* programs that crash the compiler and might be a feasible approach to minimize the exploit JavaScript to cause a crash. However, the effectiveness of this approach is unknown, because of the complex and sophisticated nature of JavaScript. Attackers can insert arbitrary junk code to make delta debugging ineffective. In contrast, JScalpel can precisely pinpoint the JavaScript statements that cause a crash and locate the malicious payload and our experiment has proven its effectiveness.

2.3 JScalpel– Overview

Figure 2 presents the architecture of JScalpel, which leverages Virtual Machine Monitor (VMM) based analysis. It consists of multiple components. A multi-level tracer is used to gather JavaScript and binary traces. A CFI module is used to determine the binary level "slicing sources", which are the violations that cause the exploit along with the various payload components. The multi-level slicer augments JavaScript level slicing with information from binary level slicing to obtain the relevant exploit and payload statements. Finally, JScalpel packages the relevant exploit statements within an HTML page to generate the minimized script. It also replaces the malicious payload with a benign payload to generate a PoV.

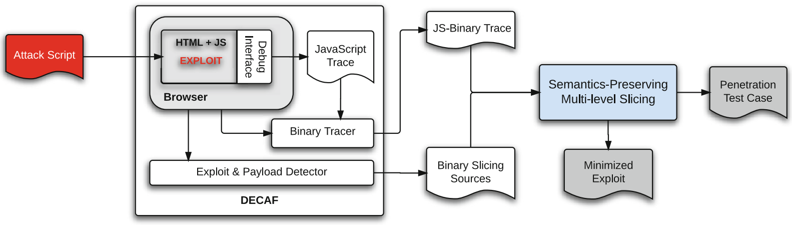


Fig. 2. Architecture of JScalpel

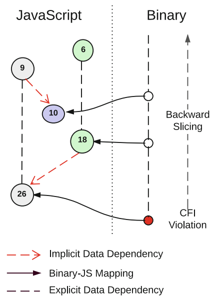


Fig. 3. Performance index of 2-relays system

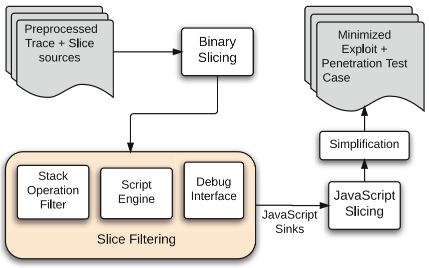


Fig. 4. Semantics-preserving multi-level slicing.

3 Multi-level Tracing and Slicing-Source Identification

We implement JScalpel on top of DECAF [19], a whole-system dynamic analysis framework. The tracing consists of two parts, JavaScript and binary tracers. JavaScript tracing is performed using a helper module that is injected into the browser address space. It interacts with the JavaScript debug interface within the browser to gather the JavaScript-level trace. The binary tracer and the exploit detection module are implemented as 2 plugins of DECAF. Below, we detail each of the components.

3.1 Context-Aware Multi-level Tracing

JavaScript Tracer. Prior approaches that gather JavaScript trace [11,21] modify JavaScript engine or the browser to identify the precise statements being executed, however such an approach requires access to JavaScript engine (and/or browser) source code which is not available for close sourced browsers like IE.

We take a JavaScript debugger-based approach. Our approach has two key advantages. (1) Most browsers – open-sourced or otherwise – support a debugging interface to debug the JavaScript statements being executed, and (2) Because the debugger runs within the browser context, it readily provides the JavaScript-level semantics. That is, we can not only gather the exact statements being executed, but also retrieve the values of variables and arguments at various statements. From within the VMM, we hook the JavaScript debugger at specific APIs to retrieve the various JavaScript statements and the corresponding contexts. The accumulation of the JavaScript statements yields the JavaScript trace.

JavaScript tracer runs as an injected module within Internet Explorer. It implements the "active script debugger" [1] interface and performs three specific actions:

1. *Establish Context*: Through the script-debugger interface, the tracer is notified when execution reaches JavaScript code. Specifically, if a `SCRIPT` tag is

encountered within an existing script or the script generated through `eval` statement, the tracer is activated with the information regarding the statement being executed. Until the next statement executes, the tracer associates the context to the current JavaScript statement.
2. *Record Trace*: At the beginning of every JavaScript statement, the tracer records the exact statement semantics along with the variable values and arguments to APIs (if any).
3. *Drive Binary Tracer*: A stub function is defined to coordinate the JavaScript tracer and the binary tracer. Before the statement executes, the binary tracer is activated along with the context information passed as the arguments of stub function such that the binary trace is associated with the particular JavaScript statement.

Binary Tracer. Binary tracer is triggered by the JavaScript tracer with the context information pertaining to a particular JavaScript statement. One way to gather a binary trace would be to monitor and capture the entire execution of the browser process at an instruction level. However, such a solution is resource intensive and inefficient. In order to be practical, our solution is selective about *what* is traced and *when* it is traced. Our goals towards an effective binary trace are to: (1) include all the relevant binary instructions that contribute to the attack, and (2) minimize the trace footprint as much as possible.

Firstly, since binary tracer is driven by JavaScript tracer, it has the precise JavaScript context. Tracing is limited and selectively turned on only when the execution is within a JavaScript statement. It is likely that the multithreading of the browser will introduce unrelated execution trace. But it does not jeopardize the analysis since all the binary instructions that contribute to the attack are included. Secondly, the effects of statements at a JavaScript-level manifest as memory reads and writes at a binary-level. Therefore, we implement a lightweight tracing mechanism. Instead of logging every binary instruction, we only log the memory read or write operations. We leverage memory IO specific callbacks supported by DECAF to record the values of *eip*, memory address, memory size, value in the memory and *esp* for each memory IO instruction. We also record the addresses of basic blocks that are executed and dump their raw bytes from virtual memory space of the monitored process at the end of every JavaScript statement. Furthermore, the binary tracer maintains information about active allocations made by the victim process. This information is used to identify self-modifying (or JIT) code. When such code is encountered, the code is dumped to the disk. When needed, the raw bytes are decoded to retrieve the actual instructions. The propagation of the slicing sources between registers and memory is identified by the memory IO logs and the binary instruction logic. While preserving the completed information as full instruction trace does for slicing process, this lightweight trace minimizes the trace size and also speeds up the slicing process.

Binary tracer is implemented as a plugin to DECAF. In the plugin, the stub function of JavaScript tracer is hooked to coordinate the binary tracing and JavaScript tracing. When the stub function is invoked by JavaScript tracer, the Binary tracer first reads the parameters of stub function from the stack where JavaScript Tracer passes the JavaScript statement and debugger information, then starts the logging of binary trace and generates a combined JS-Binary trace which contains the JavaScript and binary traces for each of the JavaScript statements. Meanwhile, a JS-binary map is built to keep track of corresponding JavaScript statement for every binary instruction.

Obfuscation and Encryption Resistance. The nature of JavaScript tracing provides inherent resistance to obfuscation and encryption because it captures each statement that is executed along with the runtime information like variable values, arguments, etc. Therefore, the intermediate statements (like the ones in Fig. 1(a)) that are used to calculate a value are each captured with their concrete values. Similarly, encrypted statements must be decrypted before they are executed, and the decrypted statements execute. Therefore, JSCALPEL encounters and records the decrypted statements that execute.

In fact, JSCALPEL performs preliminary preprocessing by performing constant folding with the help of the script execution trace. This simple optimization will not cause over simplification and generates a functionally equivalent de-obfuscated and decrypted version of the script. Then JSCALPEL executes the de-obfuscated version to perform the analysis. This preprocessing reduces the amount of analyzed JavaScript statements.

3.2 Identifying Slicing Sources

JSCALPEL makes use of a CFI module to identify slicing sources. Several solutions have been proposed to implement CFI [7]. Since JSCALPEL already relies on a VMM for trace gathering, it can leverage a VMM based CFI defense. We opt the techniques presented in Total-CFI [24] because (1) it is a recent and practical solution, (2) it has been demonstrated to work on recent real-world exploits and finally (3) it imposes low overhead.

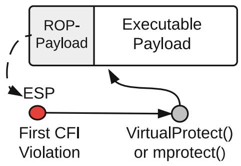


Fig. 5. Non-executable (ROP) and executable payloads used in an exploit.

It monitors the program execution at an instruction level and each point where the CFI is violated is noted as a slicing source. Albeit the recent advancement of exploitation techniques [28] can bypass the coarse-grained CFI techniques like Total-CFI, JSCALPEL's CFI module can be enhanced to include more policies to adapt the development of exploitation techniques.

Specifically, the first violation is the slicing source for the exploit-related code, whereas the subsequent violations (if any) arise from the executable payload or ROP-payload. In Fig. 5, the first violation is caused by the exploiting code, then the violations that occur up to the execution of executable payload serve as sources for ROP-payload. Moreover, the CFI module continues execution to check for executable payloads. If after the first violation, the execution ever reaches a region that within the list of allocated regions, the address is noted and it serves as the binary slicing source for the executable payload.

4 Multi-level Slicing

Multi-level slicing employed by JScalpel is based on the following hypothesis.

Hypothesis. *Implicit data dependencies at JavaScript level often manifest as direct data dependencies at binary level.*

Memory corruption exploits typically corrupt the memory by causing precise memory writes to key locations that are read by the program and result in corruption of program counter. Chen et al., show that a common characteristic of many classes of vulnerabilities is pointer taintedness [9], where a pointer is said to be tainted if the attacker input can directly or indirectly reach the program counter. In essence taint propagation reflects runtime data-flow within the program. Therefore, at a binary level, memory corruption exploits such as use-after-free, heap overflow, buffer overflow, etc. often exhibit simple data-flow, which can be captured through data-dependency analysis.

Figure 4 presents the overview of slicing employed by JScalpel. In order for the simplified exploit to be functional, it is necessary that the simplification preserves the semantics between the original and simplified scripts. Given the slicing sources and the JS-binary trace, JScalpel first performs a binary backward slice from the slice source provided by CFI violation and generates sources for JavaScript-level slicing. Slicing at the binary level ensures that no required statement is missed. Then, slicing is performed at a JavaScript level to include all the statements that sources are either data- or control-dependent on.

4.1 Binary-Level Slicing

The goal of binary slicing is to identify all the JavaScript statements that are instrumental in coercing the control flow (i.e., statements that modify the program counter) or injecting the payload into memory.

Algorithm 1 describes the backward slicing method using the lightweight binary trace. For every JavaScript statement $J[i]$, the corresponding binary instruction trace B_i is extracted. A map called "JS-Binary map" M – a mapping between the JavaScript statements and the binary instructions that execute within the statement context – is used. Then for every binary instruction $b_{ik} \in B_i$, if all of the elements in the slicing source S belong to memory locations, then the slicer checks if the current binary instruction b_{ik} has memory write operations $M_w \subseteq S$ and if it is false, the slicer jumps to the next instruction $b_{i(k+1)}$. Otherwise, the slicer does as traditional slicer to disassemble the binary instruction b_{ik} and updates the slicing source S and determine if b_{ik} should be added in the binary slice L based on the propagation rules for every X86 instruction. If L is not empty when the slicing on B_i is finished, $J[i]$ is added to the JavaScript slice O as the hidden dependency slice which may be ignored by pure JavaScript-level slicing.

In theory, a binary backward slice from the slicing sources must include all the JavaScript statements that are pertinent to the attack. However, in practice we found a key problem with such an approach. It is too permissive and ends up including *all* the JavaScript statements in the script. The main reason is the binary-level amalgamation of JavaScript and browser code along with JavaScript code. In order to track the exploit-specific information-flow, the flow through pointers must be considered. However, at a binary level, due to the complex nature of a JavaScript engine, dependencies are propagated to all the statements thereby leading to dependency explosion.

We exclude data propagation arising from code corresponding to the script engine and debug interface. Particularly, we apply the following filters to minimize the dependency explosion problem.

Algorithm 1. Binary level backward slicer

```
Input: binray trace $B$,slicing source $S$ and JS-Binary map $M$ and JavaScript trace list $J$
Output: JavaScript slice $O$
 1: $S \leftarrow \{slicing\ source\ (exploit\ point\ or\ payload\ location)\}$
 2: $O \leftarrow \emptyset$
 3: for $i = len(J); i > 0; i--$ do
 4:     $B_i \leftarrow getBinInsTraceForJS(M, J[i], B)$
 5:     for $k = len(B_i); k > 0; k--$ do
 6:         $b_{ik} \leftarrow B_i[k]$
 7:         $L \leftarrow \emptyset$
 8:         if $S$ is all memory locations then
 9:             $M_w \leftarrow GetMemWriteRec(b_{ik})$
10:             if $S \cap M_w == \emptyset$ then
11:                 continue
12:             end if
13:         end if
14:         if $getDestOperand(b_{ik}) \in S$ then
15:             $S \leftarrow S \cup updateSliceSource(b_{ik}, S)$
16:             $L \leftarrow L \cup \{b_{ik}\}$
17:         end if
18:     end for
19:     if $L! = \emptyset$ then
20:         $O \leftarrow O \cup \{J[i]\}$
21:         $L \leftarrow \emptyset$
22:     end if
23: end for
```

Stack Filtering. Once the dependency propagates to stack pointer `esp` or stack frame `ebp`, all data on the stack becomes dependent [30]. To avoid this, dependencies arising due to `esp` or `ebp` are removed during slicing. In certain cases, the stack data could be marked dependent, but when the callee returns, the dependency is discarded if it exists on a stack variable. So JSCALPEL records the current stack pointer for every read/write, and during backward slicing, when `call` instruction is encountered in the trace, the slicer checks the current stack pointer and clears the dependencies propagating from the callee's stack.

Module Filtering. During the slicing process, the propagation to or from the JavaScript engine module or script debugger is stopped. In principle, every Javascript statement executed by the same Javascript engine instance shares the data and control dependency introduced by the Javascript engine and debugger module. This kind of dependency is *outside* of "exploit specific" dependency and should be excluded from slicing.

Other Filters. Between two consecutive JavaScript statements, we found that sometimes there are data flows via CPU registers because of the deep call stack incurred by JavaScript engine and script debugger. To avoid unintended dependencies, the slicer clears the register sinks at the end of the slicing for every JavaScript statement. During our experiments (Sect. 5), we found the above filters good enough to reduce the dependency-explosion problem without missing any required statements.

4.2 JavaScript Slicing

The output of binary tracer provides the slicing sources for the JavaScript slicer. Suppose binary slice S contains n instructions. For each instruction S_i, let J_i be the JavaScript statement that represents the context under which S_i executes. Then, the JavaScript slicing sources are $\mathcal{O} = \bigcup_{i=0}^{n} J_i$. For every JavaScript statement in the slicing sources, we add the object used by this JavaScript statement to the slicing sources and include this JavaScript statement in the slice. Given the JavaScript trace, the slicer uses WALA's [4] slicing algorithm to include all the related JavaScript statement in the slice.

4.3 Minimized Exploit Script and PoV Generation

The statements are first simplified and then embedded into the exploit HTML page to obtain the minimized exploit. Also, the identified executable payload is replaced by an amicable payload to obtain a PoV in the form of a test case for the Metasploit framework.

Simplification. As a final step, JSCALPEL performs constant folding and dead-code elimination at JavaScript level to simplify the slice. It is focused on strings and constants. Specifically, for each variable v, the definitions are propagated to the uses. This is repeated for all the variables in all the statements until no more propagations are possible. Finally, if a definition of a variable has no more uses, the definition is considered dead-code and is removed only if the statement is not a source for the JavaScript slicing. This distinction is important because, the need for slice sources is already established from binary slicing. The resulting processed script is used to exploit the browser and is accepted only if the exploitation succeeds. Finally, all the statements in the script that are not a part of the slice are removed. During our experiments, we found that the simplicity of simplification incorporated by JScalpel is sufficient to bring about significant reduction in the sizes of the scripts as highlighted in Sect. 5.

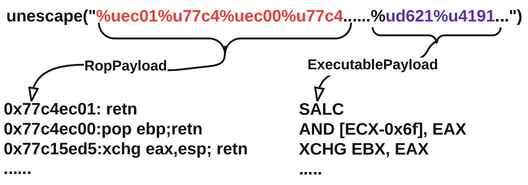


Fig. 6. CVE-2012-1876: ROP- and executable-payloads within the same string.

Collocated ROP and Executable Payloads. In some exploits, the payload and the ROP-gadgets are contained within the same string or array. For example in Fig. 6 the same string contains both ROP-payload and the executable shellcode. In such cases, JS-Binary analysis identifies the statement as both exploit and payload statement. This is an expected behavior. However, in order to replace the payload to generate the PoV, we must precisely identify the location of the start of the payload within the string. First, the JavaScript string that contains the payload is located in the memory. Then, from the payload-slice source we obtain the address of the entry point of the payload. Binary slicing from the payload-slice source leads us to the offset within the JavaScript string that corresponds to the payload. The substring beginning from the offset is replaced for PoV generation.

ROP-Only Payload. Shacham [29] showed that a set of Turing complete gadgets can be created using program text of libc. Though we cannot find any instances of *ROP-only* payload during our experiments, it is possible to compose the entire payload using only ROP-gadgets without any executable payload. Since JScalpel can locate the ROP-only payload precisely, a straightforward way is to replace malicious ROP-only payload with benign ROP-only payload.

JScalpel can generate dependent JavaScript statements in the script for any given binary-level source and the JS-Binary trace. Along with the exploit point and the payload entry point, CFI component of JScalpel captures multiple violations caused due to the ROP-gadget chain as separate binary-level slicing sources. The sources are then subject to multi-level tracing the slicing to extract the payload in JavaScript.

Disjoint Payload. Detecting the entry point of executable payload is sufficient to replace the payload and generate the PoV. However, sometimes an analyst may want to locate the *entire* executable payload. This is not a problem if the payload is allocated by the same string in the JavaScript. However, it is not necessary to be so.

JScalpel can only detect an executable payload when it executes. Therefore, it is unaware of *all* the various fragments of payload that may be injected into the memory. As a result, JScalpel will only be able to detect the JavaScript statement (and all its dependencies) that injects the entry point of the payload. It may miss some JavaScript statements that inject non-entry point payload if such statements are disjoint with the JavaScript statements that inject the entry point, and the sources for those statements are missing. Note that this is not quite a limitation for JScalpel, because the payload entry point is sufficient to generate a PoV. One way to increase the amount of payload recovered is for the CFI module to allow the payload to execute longer and capture more binary-level sources for the payload.

5 Evaluation

We evaluate JScalpel on a corpus of 15 exploits. These samples exploit the vulnerabilities discovered from 2009 to 2013 and target at Internet Explorer 6/7/8. In contrast to the large number of browser vulnerabilities discovered every year, this sample set is relatively old and small. The reasons are twofold. First, DECAF leveraged by JScalpel is based on emulator QEMU and only supports 32-bit operating system. Not all of the exploits can function correctly on DECAF. Second, it is difficult to collect working exploits although many vulnerabilities are discovered every year. We went over Internet Explorer related exploits in Metasploit, and tried to set up a working environment for each of them. We were able to set up 15 exploits on the real hardware. The remaining exploits either require specific browser/plugin versions that we were unable to find, or do not use JavaScript to launch the attacks. We then tested these 15 exploits on DECAF and 9 of them worked correctly. The 6 exploits failed to work on DECAF, because they exhibited heavy heap spray behavior, which could not finish within a reasonable amount of time in DECAF. Based on a whole-system emulator QEMU, DECAF translates a virtual memory address into its corresponding physical address completely in software implementation, and thus is much slower than the MMU (Memory Management Unit) in a real CPU. In the future, we will replace DECAF with Pin to avoid this expensive memory address translation overhead. We also crawled the Virustotal with the keyword "exploit type:html", and finally found 2 functional exploits on DECAF. In addition, from 16 exploit kits used in EkHunter [14], we managed to get 4 functional exploits from exploitkit, Siberia and Crimepack. As a result, our testset includes 9 exploits from Metasploit framework, 4 exploits from 3 different exploit kits and 2 wild exploits.

To identify the CVE number of exploits from exploit kits and wild, we ran JScalpel to extract exploitation component first and then manually searched Metasploit database and National Vulnerability Database [3] for a match. While CVE-2012-1889 exploits the vulnerability in `msxml.dll`, all the remaining samples exploit `mshtml.dll`.

Though we evaluated JScalpel on Internet Explorer only, potentially it can work on other browsers or any other programs (e.g., Adobe Reader) that have JavaScript debug interface. The experiments were performed on a server running

Ubuntu 12.04 on 32 core Intel Xeon(R) 2 GHz CPU and 128 GB memory. The code comprises of 890 lines of Python, 2300 lines of Java and 4000 lines of C++.

5.1 Minimizing Exploits

Table 1 presents the results for exploit analysis. Given one exploit, we first ran JScalpel to get the multi-level trace and CFI violation point. Then multi-level slicing was conducted to yield exploitation component and payload injection component. Based on this knowledge, our experiments demonstrate that for each exploit, JScalpel was able to generate a simplified exploit and PoV which were able to successfully exploit the vulnerability and launch the payload.

Exploitation Analysis. The binary-level slicing was conducted on the multi-level trace starting from the CFI violation point. It mapped binary level slicing results to JavaScript statements with the help of JS-binary map. The number of JavaScript statements identified by binary-level analysis is listed in Column I.

Table 1. Exploit analysis results

Source	CVE	Exploitation component					Payload injection	Simplified exploit			
		I	II	III	IV	V	VI	VII	VIII	IX	X
Metasploit	2009-0075	9	6	✓	17	✓	14	30	30	0.00	
	2010-0249	3	6	✗	19	✓	10	45	22	0.51	b c
	2010-0806	2	10	✓	10	✓	14	803	13	0.98	a c b
	2010-3962	1	1	✓	1	✓	15	105	17	0.83	a c b
	2012-1876	32	1	✗	30	✓	14	67	47	0.30	a c b
	2012-1889	1	2	✓	2	✓	67	77	77	0.00	
	2012-4969	16	1	✗	8	✓	53	117	70	0.40	b c
	2013-3163	9	1	✗	13	✓	32	43	42	0.02	a b d
	2013-3897	26	1	✗	41	✓	23	187	63	0.66	d
Wild	2011-1255	40	1	✗	16	✓	26	97	44	0.55	a b e
	2012-1889	1	2	✓	2	✓	27	53	12	0.77	a b e
exploitkit	2010-0806	2	6	✓	6	✓	13	109	29	0.73	b c
Siberia	2010-0806	2	6	✓	6	✓	12	103	22	0.79	a b c
Crimepack	2010-0806	2	1	✗	6	✓	11	198	30	0.85	a b c
	2009-0075	4	6	✗	12	✓	12	36	33	0.08	a b c

I. # of JS slicing sources.
II. # of stmts from JS analysis only.
III. Can stmts from JS-only analysis cause crash?
IV. # of stmts from JS-Bin analysis
V. Can stmts from JS-Bin analysis cause crash?
VI. # of stmts from JS-Bin analysis
VII. # of unique JS stmts of original exploit.
VIII. # of unique JS stmts of simplified exploit
IX. potency of minimization.
X. Obfuscation & fingerprinting Techniques. (^{a}: Randomization Obf. ^{b}: Data Obf. ^{c}: Encoding Obf. ^{d}: Logic Structure Obf. ^{e}: Fingerprinting tech)

They were used as the slicing sources for JavaScript level slicing. This multi-level slicing extracted the exploitation related statements the number of which were listed in Column IV. Column V shows if the extracted statements can crash the browser. For the exploits with the same CVE number like CVE-2009-0075 and CVE-2010-0806, the results of Column IV can be different due to the different implementation of exploitation. But we can see that for all of the exploits, the extracted statements can crash the browser, meaning that the semantics of exploitation component are preserved.

In comparison, the JavaScript-level only analysis cannot achieve this as presented in Column II and III. Column II lists the number of JavaScript statements obtained from backward slicing only at the JavaScript level starting from the statement that causes the first CFI violation. Column III indicates if the statements extracted from JavaScript-level slicing can cause the browser to crash. We can see that for 8 out of 15 exploits, the extracted statements do not cause a crash, which means these exploits are overly simplified in these cases. For the exploits with the same CVE number like CVE-2010-0806 and CVE-2009-0075, the JavaScript-level only analysis results were different, because the different obfuscation techniques used in these exploits introduced or eliminated unexpected dependency at JavaScript level.

Payload Injection. The CFI violation information provides the exact location of the payload in memory. The multi-level slicing yields the payload injection statements of which the number is listed in Column VI of Table 1. Column 3 in Table 2 lists the payload definition statements. For each of the exploit, our JS-Binary analysis was able to precisely pinpoint the payload injection statements for PoV generation. By contrast, solutions like JSGuard [16] or NOZZLE [26] cannot do the same, because they lack the JavaScript context and can only pinpoint the payload in the memory. Solutions by scanning the exploit code directly cannot always identify the correct payload injection statements since the payload is often obfuscated.

Minimized Exploit. For each of the exploits, we combined the payload injection statements (Column VI) with the exploitation component (Column IV) to generate a minimized working exploit. In the experiment, we observed that each minimized exploit was indeed functional, meaning that it can exploit the vulnerability and launch the payload successfully. The Column VII lists the number of unique JavaScript statements observed at the execution of the original exploit. Column VIII lists the number of unique JavaScript statements observed in the execution of the minimized exploit.

The minimized exploit excludes the JavaScript statements that belong to obfuscation code or fingerprinting code. We characterize those codes in Column X of Table 1. They cover different obfuscation or fingerprinting techniques. These techniques are designed to bypass the detection tool and make the analysis challenging. So the minimized exploit can ease the manual analysis process by removing these JavaScript statements. To quantify the degree of code complexity reduction in these minimized exploits, we adopt a metric called "potency of minimization" from an existing work [10]. A minimization is potent if it makes the

Table 2. Payload analysis results. All exploits provide a single JavaScript statement from the binary perspective, which is the context in which the exploiting instruction executes.

Source	CVE	Payload definition stmt	I	II
Metasploit	2009-0075	var shellcode = unescape("%u1c35 %u90a8 %u3abf...")	3024	✗
	2010-0249	var LLVc = unescape("%u1c35 %u90a8 %u3abf%u..")	3024	✗
	2010-0806	var wd$ = unescape((function(){return "%u4772 %u9314 %u9815..."}))	3072	✗
	2010-3962	var shellcode = unescape("%u0c74 %ud513 %uf...")	3072	✗
	2012-1876	for (var a3d = unescape("*%uec01 %u77c4 %u...*"),...)	3072	✓
	2012-1889	var code = unescape("%uba92 %u91b5 %ub0b1...")	3072	✗
	2012-4969	var GBvB = unescape("%uc481 %uf254 %uffff...")	618	✗
	2013-3163	p += unescape("*%ub860 %u77c3 %ud038...*")	36696	✓
	2013-3897	sprayHeap({shellcode:unescape("*%u868a%u77c3...*"})	696	✓
Wild	2011-1255	var sc = unescape("%u9090 %u9090 %u9090 %u1c3...")	3024	✗
	2012-1889	var mmmbc=("Data5756Data3352Data64c9...)	2880	✗
Exploitkit	2010-0806	var qq = unescape("%ucddb%u74d9 %uf424 %u...")	649	✗
Siberia	2010-0806	var qq = unescape("!5350!5251!..".replace(...")	1750	✗
Crimepack	2010-0806	var rktchpv= unescape("%u06b8 %u5c67 %udae4...")	648	✗
	2009-0075	var ysazuzbwzdqlr=unescape("%u06b8 %u5c67 %u...")	648	✗

I. Payload Length II. Collocated payload?

minimized program P' less obscure (or complex or unreadable) than the original program P. we choose the number of unique JavaScript statements observed in the execution as the metric because it represents the number of inspected statements by an analyst. This is formalized in the following definition:

Definition 1 (Potency of Minimization). Let $U(P)$ be the number of unique JavaScript statements observed at the execution of P. τ_{pot}, the minimization potency with respect to program P, is a measure of the extent to which the minimization reduces the obscurity of P. It is defined as

$$\tau_{pot} \overset{\text{def}}{=} 1 - \frac{U(P')}{U(P)}.$$

On average, the minimization potency was 0.498, which means we were able to eliminate 49.8 % of statements in the trace, whereas the maximum is 0.98. The potency of minimization of CVE-2009-0075 and CVE-2012-1889 from Metasploit are both 0, because no obfuscation techniques are applied to them. We did observe that for the exploits from the wild and exploit kits, the average potency of minimization (0.63) was higher than that (0.41) for the exploits from Metasploit. This means that it is generally more difficult to analyze the real world exploits.

5.2 PoV Generation

PoV generation is an end result of payload analysis. By replacing the payload in the minimized exploit with a benign one, a PoV is generated for penetration test. Column 3 in Table 2 lists the payload definition statements, where the payload content is first introduced or defined in the JavaScript code. The definition statement is usually accompanied with other statements required to inject the

Table 3. Effects of filtering on exploit analysis.

Source	CVE	Unique # JS stmts	# JS after pre-processing	No filter	Stack filter	Module filter	All filters
Metasploit	2009-0075	30	30	30	14	28	9
	2010-0249	45	32	32	4	32	3
	2010-0806	803	27	27	13	27	2
	2010-3962	105	17	16	16	16	1
	2012-1876	67	51	50	41	50	32
	2012-1889	77	78	78	2	77	1
	2012-4969	117	77	77	16	75	16
	2013-3163	43	43	41	4	41	9
	2013-3897	187	64	64	26	64	26
Wild	2011-1255	97	66	66	45	66	40
	2012-1889	53	53	51	1	1	1
Exploitkit	2010-0806	109	32	31	31	31	2
Siberia	2010-0806	103	27	26	26	26	2
Crimepack	2010-0806	198	195	194	22	194	2
	2009-0075	36	35	25	5	5	4

payload. Payload length (Column 4 in Table 2) is the size of the payload that was identified. In one of the samples (CVE-2013–3163), the encoder was embedded within the payload and therefore, the size of the payload was much larger than other exploits. In 3 out of 15 exploits, we found the ROP and executable payloads to be collocated within the same string. In each exploit, the payload was replaced with a benign payload and a PoV was generated.

5.3 Effects of Filtering

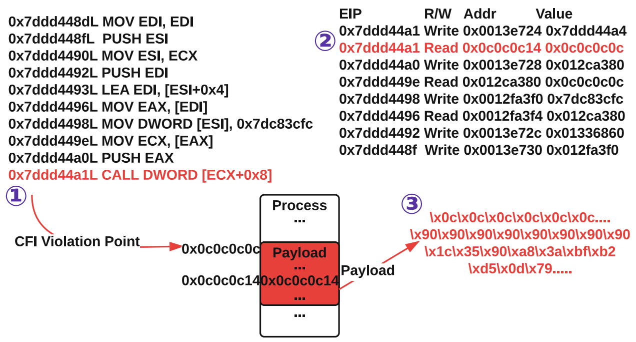
The filters help to exclude the unexpected dependencies. In Table 3, we evaluated the effects of filtering on minimizing exploits. We found that preprocessing is effective in cases where the scripts are obfuscated because, during obfuscation, multiple statements are used to accomplish the tasks of a single statement like `eval`. Column 3–4 lists the number of the unique JavaScript statements in the slicing results under different filter configurations. With no filters, we did not find any significant reduction in the slicing results. This emphasizes the need for filtering. Stack Filter and Module Filter individually produced varying amount of size reduction depending on the exploit, but in general, the combination proved to be most effective. For example, for CVE-2010-3962, the combination of all the filters reduced the number of statements to a single statement, while none of the filters were individually effective.

5.4 Case Study – CVE-2011-1255

In order to highlight the advantages of JSCALPEL, we perform a study of the wild exploit, CVE-2011-1255 [2], which exploits a "Time Element Memory Corruption Vulnerability" of the Timed Interactive Multimedia Extension implementation in Microsoft Internet Explorer 6 through 8. The exploit (MD5:016c280b8f1f155 80f89d058dc5102bb) targets Internet Explorer 6 on Windows XP SP3. Given the exploit sample, JSCALPEL successfully generated the minimized exploit code, payload injection code and penetration test template for Metasploit. We would like to highlight that a sample for CVE-2011-1255 was previously unavailable on Metasploit DB and JSCALPEL was able to generate one.

Simplified Exploit Statements JSCALPEL loads the simplified page and logs the JS-Binary trace until the CFI violation-point (detailed in Fig. 7) is reached. The violation point ① represents the hijacked control flow transfer from `0x7ddd44a1` to the payload location `0x0c0c0c0c` through an indirect call instruction – `call DWORD[ecx+0X8]`. Note that the exploit does not contain any ROP-gadgets and that the entire payload is executable. From the violation, either `ecx` or `[ecx + 0x8]` may be manipulated by the attacker and therefore both will have to be considered as possible slicing sources. From the memory IO log (point ②), the location of `[ecx+0x8]` is extracted as `0x0c0c0c14`. Both `ecx` and the memory location `0x0c0c0c14` are provided as the slicing sources for the binary-level slicer to uncover the implicit data dependency pertaining to the exploit.

The binary level slicer identified 40 JavaScript level sources. JavaScript slicer included an additional 64 statements to generate the simplified exploit. Using the simplified exploit), we were able to trigger the vulnerability in IE 6.



```
0x7ddd448dL MOV EDI, EDI
0x7ddd448fL  PUSH ESI
0x7ddd4490L MOV ESI, ECX
0x7ddd4492L PUSH EDI
0x7ddd4493L LEA EDI, [ESI+0x4]
0x7ddd4496L MOV EAX, [EDI]
0x7ddd4498L MOV DWORD [ESI], 0x7dc83cfc
0x7ddd449eL MOV ECX, [EAX]
0x7ddd44a0L PUSH EAX
0x7ddd44a1L CALL DWORD [ECX+0x8]
```

```
EIP         R/W   Addr       Value
0x7ddd44a1 Write 0x0013e724 0x7ddd44a4
0x7ddd44a1 Read  0x0c0c0c14 0x0c0c0c0c
0x7ddd44a0 Write 0x0013e728 0x012ca380
0x7ddd449e Read  0x012ca380 0x0c0c0c0c
0x7ddd4498 Write 0x0012fa3f0 0x7dc83cfc
0x7ddd4496 Read  0x0012fa3f4 0x012ca380
0x7ddd4492 Write 0x0013e72c 0x01336860
0x7ddd448f  Write 0x0013e730 0x012fa3f0
```

```
\x0c\x0c\x0c\x0c\x0c\x0c....
\x90\x90\x90\x90\x90\x90\x90\x90
\x1c\x35\x90\xa8\x3a\xbf\xb2
\xd5\x0d\x79.....
```

Fig. 7. CFI violation point

Simplified Payload-Injection Statements and Payload Location Similar to simplifying the exploit statements, JSCALPEL uses payload location *0x0c0c0c0c* as the slicing source for identifying the payload-injection statements, and gathers the simplified statements. The binary-level slicer confirmed the statement `36: a[i] = lh.substr(0, lh.length)` as the JavaScript statement that injects payload into memory. Then, this statement was used as the slicing source for JavaScript-level slicer. Finally, JScalpel identified all the payload injection JavaScript statements.)

The payload is located at `0x0c0c0c0c`. Therefore, JScalpel extracts the page at `0x0c0c0c0c` to analyze the payload. JScalpel first trims the padding instruction like `nop` from the payload. Next, JScalpel compares it with the constant strings in the payload injection JavaScript statements to identify the exact payload string. JScalpel identified (`var sc = unescape(‘‘%u9090 %u9090 %u90 90 %u9090 %u1c35 %u90a8 %u3abf%ub2d5....’’)`) as the JavaScript statement containing the payload. Since the entire payload is executable, JScalpel replaced the entire payload to generate the Metasploit test case. We generate a Ruby template script) for Metasploit framework, and we were able to successfully test it on Internet Explorer 6 on Windows XP SP3.

6 Discussion

Vulnerabilities Within Filtered Modules. If the vulnerability exploited exists within the filtered modules, the slicer produces the incomplete slice. Current implementation of JScalpel can not detect exploits that target the filtered modules. In the future, fine-grained analysis can be applied on these modules to determine which part of the code introduces the dependency and then limit the filter from whole module to some specific code range. This will reduce the number of vulnerabilities that JScalpel cannot handle.

Debug-Resistant JavaScript. In order for JScalpel to be able to analyze a script, it is important that JScalpel executes the program and monitors from the debugger. Though we did not find any samples that can detect debuggers, it is possible that exploits could use techniques (e.g., timing-based) to determine if a debugger is running and hide the malicious behavior. Currently, JScalpel is vulnerable to such attacks. It would be an interesting future work to reconstruct JavaScript-level semantics directly from the Virtual Machine Monitor, similar to how DroidScope [35]) recovers Java/Dalvik level semantic view.

Impact of JIT-Enabled JavaScript Engine on JScalpel. When JIT is enabled on JavaScript engine, the data flow within JavaScript engine becomes more complex because of the mixture of code and data. JScalpel may not work in this case. Since JScalpel is designed as an analysis tool and is not performance sensitive, the analyst can simply disable the JIT engine. However, this workaround would sacrifice the capability of analyzing attacks that perform JIT spray, as these attacks rely on the side-effects of the JIT compiler. We leave it as future work to address this issue.

7 Related Work

Drive-by-download Attacks. The drive-by-download attacks drive the emergence of “Exploit-as-a-Service” paradigm on the malware ecosystem [15]. Machine learning based approaches [6,11,13,25,32] and honeypot based approach [33] for large scale analysis have been explored to detect the malicious web pages. JShield [8] proposed a vulnerability-based approach, which uses opcode vulnerability signature to match drive-by-download attacks. NOZZLE [26] detects the existence of shellcode to identify heap spray attacks launched by malicious web pages. ZOZZLE [12] uses Bayesian classification of hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that are highly predictive of malware. BLADE [22] focuses on the client side approach by preventing unconsented content execution, which is the ultimate goal of drive-by-download attacks.

Exploit Diagnosis. PointerScope [38] uses type inference on binary execution to detect the pointer misuses induced by an exploit. ShellOS [31] built a hardware virtualization based platform for fast detection and forensic analysis of code injection attacks. Dynamic taint analysis [23] keeps track of the data dependency originated from untrusted user input at the instruction level, and detects an exploit on a dangerous use of a tainted input. explored whole system taint tracking for malware analysis. Chen et al., [9] showed that pointer taintedness analysis can expose different classes of security vulnerabilities, such as format string, heap corruption, and buffer overflow vulnerabilities. pinpoints the guilty bytes in polymorphic buffer overflows on heap or stack by tagging data from network with an age stamp. However, it is not feasible for complex attacks launched using JavaScript code.

Malicious JavaScript Analysis. To deobfuscate malicious JavaScript, Kolbitsch et al., [20] uncover environment-specific malware by exploring multiple execution paths within a single execution. Previous work [11,17,21] execute JavaScript using an emulated JavaScript running environment and acquire deobfuscated JavaScript. Our solution adopts the real browser environment and can defend most of the obfuscation techniques. JSGuard [16] proposed a methodology to detect JS shellcode that fully uses JS code execution environment information with low false negative and false positive. [21] simplify the obfuscated JavaScript code by preserving the semantics of the observational equivalence. However, the simplified JavaScript code may not exploit the vulnerability of web browser due to oversimplification. Our combined analysis can identify the JavaScript code contributing to exploit and avoid over simplification.

8 Conclusion

We presented JSCALPEL, a framework that combines JavaScript and binary analyses to analyze JavaScript exploits. Our multi-level tracing bridges the semantic gap between the JavaScript level and binary level to perform dynamic

JS-Binary analysis. We analyzed 15 JavaScript exploits, 9 memory corruption exploits from Metasploit , 4 exploits from 3 exploit kits and 2 exploits from the wild and successfully recover the payload and a minimized exploit for each of the exploits.

Acknowledgments. We would like to thank anonymous reviewers and our shepherd Dr. Manuel Egele for their insightful feedback. This research was supported in part by National Science Foundation Grant #1054605, Air Force Research Lab Grant #FA8750-15-2-0106, and DARPA CGC Grant #FA8750-14-C-0118. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. Active script debugging overview. http://msdn.microsoft.com/en-us/library/z537xb90(v=vs.94).aspx
2. Detailed analysis exp/20111255-a. http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Exp~20111255-A/detailed-analysis.aspx
3. National vulnerability database. https://nvd.nist.gov/
4. The T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/
5. Internet security threat report. https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf, April 2015
6. Borgolte, K., Kruegel, C., Vigna, G.: Delta: automatic identification of unknown web-based infection campaigns. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (2013)
7. Burow, N., Carr, S.A., Brunthaler, S., Payer, M., Nash, J., Larsen, P., Franz, M.: Control-flow integrity: precision, security, and performance. arXiv preprint (2016). arXiv:1602.04056
8. Cao, Y., Pan, X., Chen, Y., Zhuge, J.: Jshield: towards real-time and vulnerability-based detection of polluted drive-by download attacks. In: Proceedings of Annual Computer Security Applications Conference (ACSAC) (2014)
9. Chen, S., Pattabiraman, K., Kalbarczyk, Z., Iyer, R.K.: Formal reasoning of various categories of widely exploited security vulnerabilities using pointer taintedness semantics. In: Security and Protection in Information Processing Systems (2004)
10. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand (1997)
11. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proceedings of the 19th International Conference on World Wide Web (2010)
12. Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C.: Zozzle: fast and precise in-browser JavaScript malware detection. In: USENIX Security Symposium (2011)
13. Eshete, B.: Effective analysis, characterization, and detection of malicious web pages. In: Proceedings of the 22nd International Conference on World Wide Web Companion, International World Wide Web Conferences Steering Committee (2013)
14. Eshete, B., Alhuzhali, A., Monshizadeh, M., Porras, P., Yegneswaran, V.: Ekhunter: a counter-offensive toolkit for exploit kit infiltration. In: Proceedings of the 22nd Annual Network and Distributed System Security Symposium, February 2015

15. Grier, C., Ballard, L., Caballero, J., Chachra, N., Dietrich, C.J., Levchenko, K., Mavrommatis, P., McCoy, D., Nappa, A., Pitsillidis, A., Provos, N., Rafique, M.Z., Rajab, M.A., Rossow, C., Thomas, K., Paxson, V., Savage, S., Voelker, G.M.: Manufacturing compromise: the emergence of exploit-as-a-service. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (2012)
16. Gu, B., Zhang, W., Bai, X., Champion, A.C., Qin, F., Xuan, D.: Jsguard: shellcode detection in JavaScript. In: Security and Privacy in Communication Networks (2013)
17. Hartstein, B.: Jsunpack: an automatic JavaScript unpacker. In: ShmooCon Convention (2009)
18. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: tracking information flow in JavaScript and its APIs. In: Proceedings 29th ACM Symposium on Applied Computing (2014)
19. Henderson, A., Prakash, A., Yan, L.K., Hu, X., Wang, X., Zhou, R., Yin, H.: Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis (2014)
20. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: de-cloaking internet malware. In: 2012 IEEE Symposium on Security and Privacy (SP) (2012)
21. Lu, G., Debray, S.: Automatic simplification of obfuscated JavaScript code: a semantics-based approach. In: Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability (2012)
22. Lu, L., Yegneswaran, V., Porras, P., Lee, W.: Blade: an attack-agnostic approach for preventing drive-by malware infections. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (2010)
23. Newsome, J., Song, D.: Dynamic taint analysis: automatic detection, analysis, and signature generation of exploit attacks on commodity software. In: Proceedings of the Network and Distributed Systems Security Symposium, February 2005
24. Prakash, A., Yin, H., Liang, Z.: Enforcing system-wide control flow integrity for exploit detection and diagnosis. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (2013)
25. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N., et al.: The ghost in the browser analysis of web-based malware. In: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets (2007)
26. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: a defense against heap-spraying code injection attacks. In: Proceedings of the Usenix Security Symposium (2009)
27. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: 2010 IEEE Symposium on Security and Privacy (SP) (2010)
28. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., Holz, T.: Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications. In: 2015 IEEE Symposium on Security and Privacy (SP). IEEE (2015)
29. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the X86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)
30. Slowinska, A., Bos, H.: Pointless tainting? evaluating the practicality of pointer tainting. In: Proceedings of the 4th ACM European Conference on Computer systems. ACM (2009)

31. Snow, K.Z., Krishnan, S., Monrose, F., Provos, N.: Shellos: enabling fast detection and forensic analysis of code injection attacks. In: USENIX Security Symposium (2011)
32. Stringhini, G., Kruegel, C., Vigna, G.: Shady paths: leveraging surfing crowds to detect malicious web pages. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (2013)
33. Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated web patrol with strider honeymonkeys: finding web sites that exploit browser vulnerabilities. In: Proceedings of the 2006 Network and Distributed System Security Symposium (2006)
34. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. IEEE Press (1981)
35. Yan, L.K., Yin, H.: Droidscope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium (2012)
36. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, New York, NY, USA (2007)
37. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002)
38. Zhang, M., Prakash, A., Li, X., Liang, Z., Yin, H.: Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. In: Proceedings of 19th Annual Network & Distributed System Security Symposium (2012)