JSGuard: Shellcode Detection in JavaScript

Boxuan Gu[1], Wenbin Zhang[1], Xiaole Bai[2],
Adam C. Champion[1], Feng Qin[1], and Dong Xuan[1]

[1] Department of Computer Science and Engineering,
The Ohio State University, Columbus, OH, 43202, USA
{gub,zhangwen,champion,qin,xuan}@cse.osu.edu
[2] Alliance Data System, Columbus, OH, 43202, USA
alan.bai@alliancedata.com

Abstract. JavaScript (JS) based shellcode injections are among the most dangerous attacks to computer systems. Existing approaches have various limitations in detecting such attacks. In this paper, we propose a new detection methodology that overcomes these limitations by fully using JS code execution environment information. We leverage this information and create a virtual execution environment where shellcodes' real behavior can be precisely monitored and detection redundancy can be reduced. Following this methodology, we implement *JSGuard*, a prototype malicious JS code detection system in Debian Linux with kernel version 2.6.26. Our extensive experiments show that JSGuard reports very few false positives and false negatives with acceptable overhead.

Key words: malicious JavaScript code, shellcode detection, web security, intrusion detection, browser security

1 Introduction

JavaScript (JS) is a scripting language that is widely used to enrich the functionality of client-side applications, e.g., Web browsers and Adobe Reader. Unfortunately, the user experience improvement brought by JS is often accompanied by security risks since JS codes can programmatically access these applications' computational objects. There are several types of JS based attacks against client-side applications [20, 40, 41], the most dangerous of which exploits target processes' memory errors using shellcodes. Shellcodes are segments of executable codes that are injected into vulnerable processes' address spaces. After the shellcodes are injected and the control flow transfers to them, attackers can execute arbitrary code in the target hosts that can steal sensitive information, furtively download and activate malware, and carry out other nefarious tasks.

A typical example of JS based shellcode injection attacks is exploiting Microsoft Internet Explorer's (IE's) *HTML object memory corruption vulnerability* [53] using an HTML document with a specially crafted JS code embedded. After IE loads the document, the JS code is parsed, compiled, and then executed, which creates large objects containing shellcodes in IE's heap via *heap spraying* [47]. The shellcodes are activated once IE's control flow is hijacked and redirected to them.

Recently, such JS based shellcode injection attacks are growing increasingly severe [14, 40]. This stems from 2 facts: (1) users do not update their Web browsers in a timely manner yet spend more and more time surfing the Internet [21]; and (2) numerous browser plug-ins have been released, many of which have vulnerabilities [45]. The deteriorating situation is also witnessed by the popularity of "drive-by download" attacks [41] where users are duped into downloading JS codes that dynamically generate shellcodes and activate them via client-side vulnerabilities.

Unfortunately, existing solutions that detect JS based shellcode injection attacks are insufficient. Some approaches can miss detecting shellcodes since these approaches do not capture the accurate program execution environment, which is required for exposing the malicious features of shellcodes. Some approaches cannot effectively handle attacks in which shellcode is divided into several parts that are connected using control-flow-redirection instructions (e.g., `jmp`). We will present 2 representative examples in §2 that illustrate the limitations of existing detection approaches. We provide a review of existing solutions in §6.

In this paper, we focus on detecting malicious JS codes that inject shellcode into target applications. We propose a detection system that effectively overcomes the problems of existing solutions. Similar to existing work, we assume that the JS interpreter does not have exploitable memory errors and that such exploitable errors exist in the application (e.g., the Web browser) that runs the JS interpreter, plug-ins, or extension modules. We also assume that the application and its plug-ins and extensions are not malware. Therefore, we target malicious codes coming from external untrustable sources. Although we use a Web browser as an exemplary client-side target application in the rest of this paper and our prototype system is also built within a Web browser, our system can be extended to protect other client-side applications such as Adobe Reader.

To the best of our knowledge, our system is the first that creates an emulation environment *within the target application process's address space that shadows the address space information during emulation to detect malicious shellcodes in JS codes.* We perform such shadowing only when necessary. Our system accurately and comprehensively captures customized application information and real-time memory information at runtime in a lightweight manner; stand-alone machine simulators cannot easily obtain this information. From extensive experiments, we find that JSGuard yields very few false negatives and false positives. These results illustrate the promise of our detection methodology. In particular, we make the following contributions:

– We propose a new methodology that can comprehensively detect shellcodes in JS code. We propose leveraging the JS code execution environment information to instantiate a lightweight emulation environment that reveals and monitors shellcodes' real behaviors. Our emulation environment also enables examination of invoked system calls and their parameters as well as the execution flow to detect malicious shellcodes.

– We propose a technique for reducing detection redundancy at multiple levels. We fully utilize JS code execution environment information to reduce the number of times the detection system is activated and a JS string is checked.

This information includes native methods, stack frames, and properties of each individual JS object.

– We implement JSGuard, a prototype system using the above methodology in Debian Linux with kernel version 2.6.26. We integrate JSGuard into the Firefox 4 Web browser. Our system is adaptive and extensible. It is designed to run in the target process's address space. JSGuard can efficiently fetch and use JS code execution environment information for shellcode detection.

– We conduct extensive experiments based on real traces and thousands of malicious shellcode samples. The experimental results show that our malicious JS code detector has high detection accuracy with acceptable overhead.

Paper Organization. The rest of this paper is organized as follows. §2 provides background information and motivating examples. §3 presents our system design and implementation. §4 presents detection examples. §5 evaluates JSGuard's performance. §6 reviews related works. §7 concludes.

2 Background And Motivating Examples

In this section, we provide a brief background on detecting shellcode in JS objects. Then we use 2 examples to illustrate the limitations of existing approaches.

2.1 Background: Detecting Shellcode in JS Objects

Malicious JS code usually places shellcode into objects generated at runtime and then activates it by exploiting vulnerable applications' memory errors. Therefore, detecting shellcode in JS objects is critical. Existing detection approaches can be classified into 2 categories: *content analysis* and *hijack prevention.*

Content Analysis The approaches in this category are based on scanning JS objects' contents to determine if they contain malicious shellcode. It can be further divided into 2 sub-categories: *static analysis* and *dynamic analysis.* In static analysis, input data are first disassembled and then screened via code-level pattern analysis and matching. Patterns can be complicated signatures or simple heuristics that are obtained from studying known malicious codes. A representative work is Nozzle [43]. Static analysis detection is fast, but it is known that determining program behavior via static analysis is generally undecidable and, often, it can be effectively thwarted by obfuscation techniques [5].

Dynamic analysis based methods detect malicious shellcode by exploiting information generated during shellcode execution. A representive work is [18] that uses `libemu` [28] to detect shellcode in JS strings. The state of the art of dynamic analysis is network-level emulation, which decodes input data into instruction sequences and then emulates their execution [28, 37–39]. If any of them exhibits malicious behavior during emulation, the input data are classified as malicious. Even though network-level emulation can achieve better detection completeness than static analysis, it is still prone to evasion. This is because it assumes that the working shellcodes either are self-contained or use specific memory access behaviors, i.e., their executions are independent of the dynamics of the JS code execution environment. Without knowledge of the execution environment, these approaches can be fooled by shellcode whose execution takes advantage of virtual memory information in the target process.

```
0000  6a7f      push $0x7f
0002  59        pop  %ecx
0003  6a08      push $eaddr;eaddr=0x08
0005  5e        pop  %esi
0006  46        inc  %esi
0007  4e        dec  %esi
0008  fec1      incb %cl
000a  80460ae2  addb $0xe2,0xa(%esi)
000e  304c0e0b  xorb %cl,0xb(%esi,%ecx)
0012  00fa      addb %bh,%dl
0014
.........<encrypted payload>.........
0093
```

(a)

```
0000  6a7f      push $0x7f
0002  59        pop  %ecx
0003  6a08      push $eaddr
0005  5e        pop  %esi
0006  46        inc  %esi
0007  4e        dec  %esi
0008  fec1      incb %cl
000a  80460ae2  addb $0xe2,0xa(%esi)
000e  304c0e0b  xorb %cl,0xb(%esi,%ecx)
0012  e2fa      loop 0xe
000e  304c0e0b  xorb %cl,0xb(%esi,%ecx)
0012  e2fa      loop 0xe
....  ........  .......................
```

(b)

Fig. 1. (a) Self-modifying shellcode example. The second column indicates the address of each instruction, the third column indicates the instruction binary code, and the fourth column is the IA-32 assembly code. The shellcode is mapped to address `0x0000`. (b) Execution trace of the self-modifying shellcode shown in Fig. 1(a).

Hijack Prevention As suggested by the name, hijack prevention approaches focus on preventing shellcode from being fully executed. This is often achieved by inserting special characters into the shellcode. A representative example is Bubble [22]. In Bubble, a JS string object is divided into multiple units, each 25 bytes long. In each unit, Bubble inserts `0xCC` (i.e., `int 3`) into a randomly selected position. If a JS string object contains shellcode and the shellcode is executed, an interrupt handler will be activated when the control flow reaches the insertion point. However, existing hijack prevention approaches cannot effectively detect shellcodes split into parts that are "connected" at runtime via instructions that alter control flow, e.g., `jmp` and `call`.

In the following, we first introduce the heap spraying technique. Then we present 2 examples using it that can evade content analysis and hijack prevention approaches.

2.2 Heap Spraying

Heap spraying is an attack technique to thwart address space layout randomization (ASLR) [6, 36], a memory protection mechanism where objects' positions are randomly arranged in a process's address space. ASLR intents to prevent attackers from easily predicting target object addresses. However, the memory space that can be randomized is often limited, especially in 32-bit operating systems. If we allocate many large objects in the heap, then new objects will likely be placed in a contiguous memory area after a number of allocations, making their positions predictable. This technique is called heap spraying [17, 47].

2.3 Example 1: Thwarting Content Analysis Approaches

Fig. 1(a) shows a shellcode that is modified from an example illustrated in [37]. In the shellcode, `eaddr` is used to calculate the addresses at which the encrypted payload can be accessed. Since heap spraying can make the positions of some heap objects predictable, a skilled attacker can write JS code that first sprays target processes' heaps, and then inserts the shellcode into the objects whose addresses can be predicted and determined. In this example, we assume that the starting address of the shellcode is `0x0000` and `eaddr` is `0x0008`.

This shellcode modifies its instructions at runtime. From address `0x0014` to address `0x0093`, there is an encrypted payload, which often appears to be a meaningless or invalid instruction sequence. When the control flow reaches address `0x000a`, the instruction `addb $0xe2, 0xa(%esi)` will be executed. This instruction modifies the contents of memory at address `0x0012`. After it is executed, the instruction at address `0x0012` is modified to `loop 0xe`, which forms a backward loop to decrypt instructions from `0x0093` to `0x0014`. The loop is controlled by register `ecx`, which decreases by 1 upon each execution of `loop 0xe`. Within the loop, the instruction at address `0x000e`, `xorb %cl, 0xb(%esi,%ecx)`, is for decryption. It decrypts 1 byte per iteration. When `ecx` becomes 0, the loop terminates, the content stored from `0x0093` to `0x0014` is fully decrypted, and the control flow continues to the instruction at address `0x0014`, the last decrypted instruction. We can see this from the shellcode execution trace shown in Fig. 1(b).

As there is no information that is dynamically generated during shellcode execution, e.g., register values at runtime, static analysis based detection approaches cannot effectively handle the decryption procedure after the shellcode is interpreted as an instruction sequence; these approaches only see the encrypted payload as a meaningless or invalid instruction sequence. Malicious behaviors that are only exhibited during execution are thus effectively concealed.

The shellcode shown in Fig. 1(a) can also be used to evade detection by current dynamic analysis based tools [18,28,37–39]. Given an input stream containing the shellcode shown in Fig. 1(a), network-level emulation based approaches will copy the input stream into a memory space that performs this emulation, and all read/write operations will be performed in the emulated memory space. The real contents of virtual memory units at the addresses calculated from `eaddr` are difficult to obtain. Then the shellcode's encrypted payload cannot be correctly decoded and emulated. In addition, these approaches do not use information in other objects to detect shellcode in the current object, which precludes shellcode detection. Since the use of heap spraying can enable prediction of objects' positions in a heap, it is not difficult for attackers to design shellcode in JS code that makes use of information stored in different objects. For example, if 2 JS objects have predictable heap positions, attackers can store shellcode in one and critical information for decryption in the other.

We also notice that some tools based on network-level emulation use heuristics based on the GetPC code [24, 37] in shellcode detection, e.g., [18] uses `libemu` [28]. Besides the aforementioned evasion methods, attackers can also evade detection by writing shellcode without *call group* instruction or `fstenv` instruction opcodes, e.g., using purely alphanumeric shellcode [31]. Note the shellcode shown in Fig. 1(a) has no bytes that can be decoded as the GetPC code.

2.4 Example 2: Thwarting Hijack Prevention Detection

In this subsection, we discuss how to design shellcode that evades hijack prevention detection. Fig. 2 shows a shellcode that can open a root shell. This shellcode can be divided into 3 parts as shown in Fig. 2. The first part, desub-shellcode1

```
be20010505   movl $Saddr,%esi
8976f8       movl %esi,-0x8(%esi)
836ef810     subl $0x10,-0x8(%esi)
31c0         xor %eax,%eax
eb09         jmp Offset1
```

sub-shellcode2

```
8846f7   movb %al,-0x9(%esi)
8946fc   movl %eax,-0x4(%esi)
b00b     mov  $0x0b,%al
8b5ef8   movl -0x8(%esi),%ebx
8d4ef8   leal -0x8(%esi),%ecx
8d56fc   leal -0x4(%esi),%edx
cd80     int  $0x80
eb04     jmp  Offset2
```

sub-shellcode3

```
31db   xor %ebx,%ebx
89d8   mov %ebx,%eax
40     inc %eax
cd80   int $0x80
```

Fig. 2. A shellcode can be divided into multiple parts (3 parts here). Each part, denoted by *sub-shellcode*, can be connected to another part by using a `jmp` instruction.

noted *sub-shellcode1*, is 16 bytes long. The second part, *sub-shellcode2*, is 21 bytes long. The third part, *sub-shellcode3*, is 7 bytes long. In sub-shellcode1, `Saddr` is `0x05050120` pointing to some part of an object. The memory at address `(Saddr-16)` stores arguments of the system call used to open a root shell. These include an ASCII sequence `/bin/sh`. At the end of sub-shellcode1, there is an instruction `jmp Offset1`, where `Offset1` is the offset between sub-shellcode1 and sub-shellcode2. This instruction diverts control flow from sub-shellcode1 to sub-shellcode2. In sub-shellcode2, `Offset2` is the offset between sub-shellcode2 and sub-shellcode3. At the end of sub-shellcode2, instruction `jmp Offset2` diverts control flow from sub-shellcode2 to sub-shellcode3.

Using heap spraying, the arguments and the sub-shellcodes can be placed into 2 different objects whose positions can be predicted. Let the arguments be placed in *object1* and sub-shellcode1, sub-shellcode2, and sub-shellcode3 be placed in *object2*. Because the data structures of *object1* and *object2* are known to the attacker, it is not difficult to arrange and predict the addresses of the arguments and the above 3 sub-shellcodes in memory.

Consider a Web browser with a certain memory vulnerability that can be exploited to overwrite a function pointer and thus execute arbitrary code. The attacker can use sub-shellcode1's address to overwrite the function pointer. After the web browser's control flow is directed to sub-shellcode1 and the instruction `jmp Offset1` is executed, the control flow can be directed to sub-shellcode2, and then to sub-shellcode3 through the instruction `jmp Offset2`. In this way, the entire shellcode can be executed and a root shell is opened eventually.

Existing hijack prevention approaches may fail to detecting such shellcode with high probability. For example, if the 3 sub-shellcodes are placed at the beginning of 3 25-byte blocks in *object2*, the probability that the entire shellcode can evade detection by Bubble [22] is $(25-16)/25 \times (25-21)/25 \times (25-7)/25 =$ 4.1%. This implies, on average, more than 4 attacks can succeed per 100 trials.

Example 2 also illustrates the importance of JS code execution environment information for an attack. The arguments of the system calls used by the shellcode embedded in *object2* rely on information stored in *object1*.

These examples presented in §2 clearly demonstrate the criticality of fully leveraging JS code execution environment information in order to detect shellcode in JS objects. In addition, to guarantee detection completeness, we need to check all possible instruction sequences that can be decoded.

3 System Design and Implementation

In this section, we present the design methodology of JSGuard, its architecture and key components, and implementation. The detailed workflow will be further illustrated by examples in §4.

3.1 Design Rationale

Fundamentally, the limitations of existing approaches arise because they do not fully use the JS code execution environment information during detection. This motivates our proposal of a new detection approach that overcomes the limitations by efficiently and fully exploiting this information, including: (1) the virtual memory contents of the target application running the JS interpreter; (2) the host system's context information, e.g., system call information; and (3) the JS code semantics, which include stack frames, native method information, JS object properties, etc.

This information is used at the core of JSGuard in the following 2 ways:

– *Creating a Virtual Execution Environment for Detection.* When our detection system is activated, the real environment information at that moment is used to instantiate a virtual environment where potentially malicious JS strings are executed and monitored. Such real environment information is critical for observing the real behaviors of possible shellcodes as they exhibit real execution flow. In malicious shellcodes, process state information can be used to redirect the execution flow, e.g., for encryption or decryption (as illustrated in Example 1) or it can be leveraged to compute arguments for system calls to perform malicious actions (as illustrated in Example 2). Without precise virtual memory information, the shellcode's execution flow or characteristics can be changed and its malicious behavior may not be captured.

Using the real environment information also enables leveraging a target system's binary code to emulate system calls appearing in a decoded instruction sequence, especially those that do not change processes' states but can be used to take part in shellcode computation. This kind of emulation can help us observe more possible shellcode behaviors.

– *Facilitating Multiple-level Redundancy Reduction.* We propose reducing detection overhead at 3 levels. First, the number of JS objects to be checked should be minimal. Second, given a JS object to be checked, checking occurs only as necessary (e.g., after mutable objects have changed). Finally, the detection system should be activated as infrequently as possible.

We achieve this multiple-level redundancy reduction at JSGuard's core using the following execution environment information: stack frames, properties of JS objects, and native methods. The JS interpreter maintains a stack frame for each JS function being interpreted including its origin information. By searching the current stack frames, we can determine if JS functions are internal functions or from trustable sites. If not, objects generated in JS functions are to be checked. In addition, since external components are targets of malicious code, our detection system is activated right before control flow enters them. External components are called by JS code via external native methods. Native method information is used to distinguish built-in JS native methods that are secure (as we assume

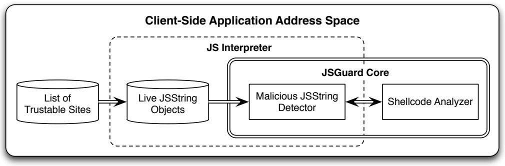

Fig. 3. The overall architecture of JSGuard.

the JS interpreter is secure) from external ones that are written by users to call their external components. We only activate our detection system before external native methods are called.

3.2 JSGuard Architecture and Key Components

JSGuard aims to detect whether JS codes embedded in webpages generate malicious shellcode. If a JS code generates such shellcode at runtime, it is considered malicious. Like other work [18,22], JSGuard focuses on detecting shellcode in JS string objects, as it is difficult to insert shellcode in other types of objects.

As illustrated in Fig. 3, JSGuard resides in the address space of the target process. Besides the *JSGuard core*, the core functionality block that performs detection, JSGuard also involves the JS interpreter and a list of trustable sites. The JS interpreter determines the origins of JS functions being interpreted; only those from external untrusted sites are further checked by the JSGuard core. The list can be maintained manually or automatically. New sites can be added to it according to JSGuard's detection results for them as well as the user's knowledge. These sites can be those that are often visited by the user, e.g., the site of the company he or she is working for. They can be also those maintained by reputable companies or organizations, such as Microsoft, CNN, etc. If users are concerned about a trustable site, they can always force JSGuard to check it. The list entries can be trustable organizations' hostnames or domain names.

As shown in Fig. 3, JSGuard core has 2 key components: the malicious JS string detector and the shellcode analyzer. The malicious JS string detector runs in the JS interpreter. It prepares JS strings to be checked at runtime and then feeds them to the shellcode analyzer. The shellcode analyzer checks if an input object's content contains malicious shellcode or a part thereof and reports the results back to the malicious JS string detector. If a malicious JS string is found, interpretation stops; otherwise, it continues. In the following, we detail these components.

Malicious JavaScript String Detector As shown in Fig. 4, the detector retrieves and checks JS strings from a checking list, which contains all JS strings that might have malicious shellcode. The checking list is maintained by instrumenting string-related operations and the JS interpreter's garbage collector (GC). In particular, when a new string JS is created, it is inserted into the checking list; when the GC reclaims a JS string, the string will be removed from the checking list after its content is zeroed.

The basic workflow of the malicious JavaScript string detector is shown in Fig. 5. The function `maliciousJSStringDetector()` has an input `checkinglist`. When called, it scans all strings in `checkinglist` and feeds them to `shellcodeAnalyzer()`, which detects malicious shellcode in JS string contents. If

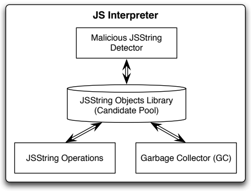
JS Interpreter
Malicious JSString Detector
JSString Objects Library (Candidate Pool)
JSString Operations
Garbage Collector (GC)

Fig. 4. Malicious JS string detector takes JS strings from a pool maintained by string-related operations and the JS interpreter's GC.

```
#define   BENIGN      0
#define   MALICIOUS   1

struct JSString {
    size_t          length;
    jschar          *chars;
};

int maliciousJSStringDetector(checkinglist) {
  JSString *string;
  check = checkinglist;
  while(check != NULL) {
    string = check->string;
    if(ShellcodeAnalyzer(string->chars)==MALICIOUS)
        return MALICIOUS;
    check=check->next;
  }
  checkinglist = NULL;
  return BENIGN;
}
```

Fig. 5. Workflow of malicious JS string detector.

shellcodeAnalyzer() finds a JS string containing malicious shellcode, it returns MALICIOUS to maliciousJSStringDetector(). Then maliciousJSStringDetector() stops checking the remaining JS strings in checkinglist and returns MALICIOUS to the JS interpreter, which stops interpreting JS code. If no JS string is found to be malicious, then maliciousJSStringDetector() returns BENIGN to the interpreter, which continues interpreting JS code. In JSGuard's core, maliciousJSStringDetector() is called immediately before JS code calls an external component.

checkinglist contains the JS strings to be checked. Every time a JS string is generated, all current stack frames are checked. If there are any JS functions from external untrusted sites, then we add the JS string to checkinglist. We do so because only JS codes from external untrusted sites attempt to generate shellcode that exploits target applications' vulnerabilities. As JS strings are immutable objects, we can safely remove the strings from checkinglist after they have been checked once [22].

Shellcode Analyzer The shellcode analyzer architecture is shown in Fig. 6. This module consists of an instruction decoder, an instruction emulator, a malicious behavior detector, an emulated memory system, and emulated registers.

Given a position in a JS string content, the instruction decoder decodes instructions starting at that position and sends each decoded instruction to the emulator. For each instruction the emulator receives, it emulates the execution thereof, for which the emulated memory system and registers provide a virtual runtime environment. The JS code execution environment information provided to the shellcode analyzer includes the target process's address space, current registers, and other context information as necessary. The emulator executes each instruction sequence and the malicious behavior detector determines whether there is any malicious behavior. If any such behavior is detected, then the instruction sequence is considered malicious. As a result, the shellcode analyzer concludes there is malicious shellcode in the content buffer. Hence the JS string object is considered malicious.

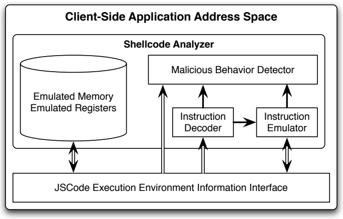
Client-Side Application Address Space
Shellcode Analyzer
Emulated Memory
Emulated Registers
Malicious Behavior Detector
Instruction Decoder
Instruction Emulator
JSCode Execution Environment Information Interface

Fig. 6. Shellcode analyzer architecture.

```
 1 #define MALICIOUS 1
 2 #define BENIGN    0
 3 #define MALICIOUS_SEQUENCE 1
 4 #define BENIGN_SEQUENCE    0
 5
 6 int ShellcodeAnalyzer(base_addr, base_size) {
 7   for (i = 0; i< base_size; i++)
 8     if (MaliciousInstructionSeq(base_addr + i))
 9       return MALICIOUS;
10   return BENIGN;
11 }
12
13 int MaliciousInstructionSeq(addr){
14   InitializeEmulationEnvironment();
15   instruction = InstructionDecoder(addr);
16   if (End(instruction)) return BENIGN_SEQUENCE;
17   instruction.exe_depth = 1;
18   while (instruction) {
19     if (MaliciousSystemCall(instruction))
20        if (instruction.exe_depth > exe_depth_thresh)
21          return MALICIOUS_SEQUENCE;
22        InstructionEmulator(instruction);
23        UpdateEmulationEnvironment();
24        target = ComputeTarget(instruction);
25        prevInstruct = instruction;
26        instruction = InstructionDecoder(target);
27        if (End(instruction)) break;
28        SetExecutionDepth(instruction, prevInstruct);
29   }
30   return BENIGN_SEQUENCE;
31 }
```

Fig. 7. Workflow of shellcode analyzer.

During instruction sequence emulation, if there is an instruction that reads memory, the memory values are first fetched from the real memory units in the target process's address space. Next, these values are stored in the emulated memory system. Future read operations to the same memory units will be directed to the emulated memory system. If there is a write memory operation, it will be directed to the emulated memory system. The write operation is never performed on the corresponding real memory units in the target process's address space to avoid disturbing "normal" JS code execution.

The shellcode analyzer workflow is shown in Fig. 7. From each input data position, the shellcode analyzer uses the target process's virtual memory information to emulate the execution of the decoded instruction sequence. There are 2 input parameters for `ShellcodeAnalyzer()`: (1) `base_address`, the starting address of the input data to be analyzed; and (2) `base_size`, the input data size.

The key function of the shellcode analyzer is `MaliciousInstructionSeq()`, which detects a malicious instruction sequence. The workflow of `MaliciousInstructionSeq()` is shown in lines 13–31 in Fig. 7. The `while` loop from line 18 to line 29 in Fig. 7 emulates a sequence of instructions, which continues until one of the following occurs: (1) a malicious behavior is detected; (2) a privileged or invalid instruction is encountered;[1] (3) an illegal memory access occurs; or (4) the number of executed instructions exceeds a threshold.

[1] Privileged instructions can only be executed in kernel mode; shellcodes normally run in user mode. An exception occurs if a shellcode contains a privileged instruction.

In our system, a *malicious behavior* is defined as a *malicious* system call invocation. In Linux and Microsoft Windows systems, not all system calls can compromise the target host's security. This depends on system call numbers and parameters, which are stored in registers before system call instructions are executed. Through the JS code execution environment information interface, the system call number and its parameters can be accurately obtained to determine if the system call invocation is intended to compromise the host's security. For example, in Linux, the system call number `11` corresponds to the system function `execve`, which executes a program. During instruction emulation, if the instruction is a system call instruction and the value of the emulated `eax` is `11`, then the system call number is `11`. After checking parameters stored in other emulated registers and the emulated memory system, if its first parameter is `/bin/sh`, then we can conclude that the instruction tries to open a root shell. In this case, the system call instruction will be considered malicious.

Shellcodes normally need several instructions to initialize system call parameters. Hence, we also use the `exe_depth` of an instruction that invokes a system call to decrease false positives. An instruction's `exe_depth` is defined as the number of instructions from the starting point to it during emulation of an instruction sequence. For example, suppose that a statement S in a `for` loop is executed 100 times. Then the execution depth of S is `2` (`for` statement and S).

Our system can also leverage heuristics used in current network-level emulation tools [18, 28, 37–39] to detect shellcode in JS strings during emulation. However, these heuristics are confined to detect particular types of shellcode that exhibit self-decrypting behavior [18, 28, 37, 38] or match specific memory access patterns [39]. In addition, as illustrated in §2, they are ineffective at detecting shellcode that fully exploits JS code execution environment information.

3.3 Implementation

The JSGuard prototype system is implemented in Debian Linux with kernel version 2.6.26 using C and C++ with `gcc` 4.3.2. The key component is the JSGuard core, which comprises 2 major parts. The first part is a modified JS interpreter integrated with the malicious JS string detector. This part is based on the SpiderMonkey JS interpreter [49], which is used in various Mozilla products including Firefox. The second part is the shellcode analyzer module. We implement it as a C library in Debian Linux system. When the malicious JS string detector calls the module, it is loaded into the address space of the application running the JS interpreter. We also implement a Firefox extension that maintains the list of trustable sites, which is loaded into Firefox's address space upon execution.

Modified JS Interpreter In this part, we implement a malicious JS string detector, which scans JS string objects from a `checkinglist` and then calls the shellcode analyzer to determine if they have malicious content. The `checkinglist` is maintained by the code that we add into all functions related to JS string operations. First, we instrument all functions related to JS string object creation. In this way, we can track all JS string objects generated during execution of the external JS code. Populating the `checkinglist` with all strings fundamentally guarantees the completeness of our detection. Second, before adding a JS string to checkinglist, we also use the list of trustable sites and current stack frames to decide if the JS string should be added to checkinglist. If all JS functions being interpreted are from trustable sites or internal JS functions, the string will not be added to checkinglist; otherwise, it will.

After analyzing the source code of the SpiderMonkey JS interpreter, we find all call points that invoke native methods and insert calls to the malicious JS string detector at these points. Since the JS interpreter also uses native methods to implement some built-in JS class methods, we check if a native call is calling a JS built-in method at native call points. If this is the case, we do not activate the malicious JS string detector; otherwise, we activate it. This is due to our assumption that the JS interpreter has no exploitable memory errors. The native methods for JS built-in class methods are parts of the JS interpreter, so they do not have exploitable memory errors. However, when control flow leaves the JS interpreter to external functions, the malicious JS string detector will be activated to check all JS strings in the checkinglist.

We modify the JS interpreter's garbage collector to maintain the checkinglist and integrate the modified JS interpreter into the Firefox 4 Web browser.

Shellcode Analyzer The shellcode analyzer prototype focuses on the IA-32 architecture and the Linux OS. We implement an instruction emulator and an instruction decoder, which is based on the Bastard project's libdisasm with version 0.23-pre [50].

When encountering a system call instruction (sysenter or int 0x80) in emulation, the shellcode analyzer will determine, with the parameters stored in the emulated memory/register system, whether it is one of 36 system calls that can be used to compromise the Linux system [32]. Besides these "malicious" system calls, we also use the exe_depth threshold to determine if the instruction truly tries to compromise the host's security; we set the threshold to 10 since most unencrypted malicious shellcodes have at least 10 instructions [38, 55]. To avoid an infinite loop during instruction sequence emulation decoded from a position of a JS string's content, we set the threshold to 8000 for the number of executed instructions. According to current research, this threshold suffices to detect malicious shellcodes [37, 38].

4 A Detection Example

We illustrate our detection system's effectiveness by presenting the detection procedure for Example 2 in §2.3. Example 1 in §2.2 can similarly be detected.

Assume that an attacker tries to exploit a Firefox external component in Linux using a malicious JS code. He first uses heap spraying to allocate large JS objects, then inserts the arguments and the 3 sub-shellcodes, as shown in Fig. 2, into 2 objects. We denote these objects as *object1* and *object2*. The objects are allocated in 2 contiguous memory areas and their addresses are predictable, say, 0x05250020 and 0x05350020, respectively. The JS code places the arguments in *object1* with Saddr set to 0x05250084 and places sub-shellcode1, sub-shellcode2 and sub-shellcode3 into *object2* with their addresses set to 0x05350084, 0x0535009D and 0x053500B6 respectively. Then the offset between sub-shellcode1 and sub-shellcode2 is 9 and the offset between sub-shellcode2 and sub-shellcode3 is 4. Hence, in Fig. 2, `Saddr` is `0x05250084`, `Offset1` is 9, and `Offset2` is 4.

The attack starts when the 3 sub-shellcodes are ready in the heap. The JS code calls the vulnerable component. Before control flow is diverted from the JS interpreter to the external component, the JS interpreter with JSGuard invokes `maliciousJSStringDetector()` to check whether there are malicious JS strings arranged in the heap. `maliciousJSStringDetector()` will scan JS strings in `checkinglist` and send them iteratively to the shellcode analyzer. At a certain moment, the shellcode analyzer receives the content of *object2*.

The shellcode analyzer decodes every possible instruction sequence starting from each byte position of the content, and then executes it. Each instruction in the instruction sequence starting from the address `0x05350084` will be decoded and then executed. When the instruction `jmp Offset1`, i.e., jumping to `0x0535009D`, is decoded and executed, the shellcode analyzer will follow the control flow and begin to decode instructions starting from `0x0535009D` and execute them. Note `0x0535009D` is the starting address of the sub-shellcode2 instruction sequence. In this way, the instruction sequence of sub-shellcode2 is discovered and executed. When system call instruction `int $0x80` is executed, we can obtain its parameters since the contents of the emulated registers/memory system precisely reflect the runtime changes during the emulation. The shellcode analyzer discovers that this system call instruction tries to open a root shell. Meanwhile, this instruction's `exe_depth` exceeds the threshold. Thus this system call instruction will be considered malicious. As a result, the entire emulated instruction sequence is considered malicious. The shellcode analyzer concludes that *object2* content contains malicious shellcode and returns to `maliciousJSStringDetector()`. When the malicious JS string detector receives `MALICIOUS` from the shellcode analyzer, it in turn concludes that *object2* is a malicious JS string and the JS code being interpreted is malicious. It throws an exception and stops interpreting JS code.

5 Evaluation

We conduct extensive experiments to evaluate JSGuard, particularly its detection effectiveness and runtime overhead. We do so on a HP Pavilion a815n with an Intel Pentium 4 3.06 GHz CPU and 1 GB RAM. The computer is connected to a university campus network through 100 Mbps Ethernet; it runs Debian Linux with kernel version 2.6.26.

5.1 Effectiveness

Detection effectiveness is measured by false positives and false negatives.

– *False Positive: 0/2000.* We implement a Firefox extension that automatically fetches websites listed in a file. We set the time interval between 2 fetches to be 50 s, which is generally sufficient for JS codes embedded in a webpage to be fully executed. Every 50 s, the extension iteratively reads a URL from the file and then loads the webpage in a browser window. We construct a benign URL list containing 2000 URLs taken from the Alexa ranking of top global sites [1]. These are real websites with various content and Web applications. JSGuard classifies all of them as benign.

– *False Negative: 0/5063.* We collect 12 real world malicious webpages containing JS code that generate shellcode to launch attacks; we also collect 51 plain malicious shellcodes from the Internet. All of them target Linux systems. Based on the 51 plain shellcodes, we use the following tools to generate 5000 polymorphic or/and metamorphic malicious shellcodes: the Metasploit project's JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai [51] as well as ADMmutate [30] and TAPiON [2], which are also used in other shellcode detection tools [37, 38, 54, 55] to test their effectiveness. We then create 5051 JS codes that generate these malicious shellcodes at runtime and invoke native methods that are not built in to the JS interpreter. For example, the JS method `document.write()` eventually calls a native method. Finally we craft 5051 malicious webpages with these malicious JS codes. We put these 5051 malicious webpages and the 12 real world malicious webpages on our internal Web server and we visit them using Firefox with JSGuard on a client computer. JSGuard classifies all of them as malicious. In addition, we also write 2 heap spraying JS codes, dynamically generate the 2 shellcode examples presented in §§2.2–2.3, and feed them to JSGuard. It correctly classifies them as malicious.

5.2 Overhead

To measure JSGuard's overhead, we use 2 versions of Firefox 4: one integrated with JSGuard and an "original" version without JSGuard. We use the 100 most popular websites as described by Alexa [1] as the testing dataset. In our experiments, we visit each website 3 times using each version of Firefox. The time we measured, *rendering time*, includes the times for downloading a webpage from the Internet, page parsing and rendering, and executing all JS codes therein.

We performed 3 types of experiments to measure overhead incurred: (1) by only checking trustable sites; (2) by only using the JSGuard core functionality block; and (3) by using entire JSGuard system.

In the first experiment, we disable JSGuard and measure the overhead purely incurred by checking trustable sites. We use the 10,000 most popular websites from Alexa [1] to form a list of trustable sites. The experiment results are shown in Table 1, which shows that this overhead is very low. Thus our detection system has little impact on the rendering time when all JS functions called during runtime are internal ones or from trustable sites. The second experiment measures the overhead purely incurred by running JSGuard core *without* checking trustable sites. It is an extreme case where every site the user visits is assumed to be malicious, i.e., every JS string is put into `checkinglist` so long as all interpreted JS functions are from external sites. From Table 2, the average overhead incurred by JSGuard core is 3.865 s. Note that this performance is measured in the worst-case scenario with a low-end machine. Indeed, studies show that overall user frustration increases when page load times exceed 8–10 s [8,33]. Hence, performance is acceptable even in this extreme case. The third experiment measures the overhead incurred by the entire JSGuard system. We construct a random list of 50 trustable sites from our testing dataset. The remaining 50 sites in our testing dataset are thus considered untrustable. Table 3 shows the experiment results. JSGuard's average overhead is modest: ∼0.87 s.

Firefox Version	**Total Time**	**Time/Page**	**Overhead/Page**
Original Version	491.953 s	1.63984 s	N/A
Trustable List Only	492.254 s	1.64085 s	0.00101 s

Table 1. The overhead of checking trustable sites only. “Original version” is Firefox without our system. “Trustable List Only” is Firefox with our detection system enabled (JSGuard core disabled).

Firefox Version	**Total Time**	**Time/Page**	**Overhead/Page**
Original Version	491.953 s	1.63984 s	N/A
JSGuard Core Only	1651.45 s	5.50483 s	3.86499 s

Table 2. The overhead purely incurred by the JSGuard core block. “JSGuard Core Only” is Firefox with our system enabled (checking trustable sites disabled).

Firefox Version	**Total Time**	**Time/Page**	**Overhead/Page**
Original Version	491.953 s	1.63984 s	N/A
With JSGuard	753.059 s	2.51019 s	0.87035 s

Table 3. The overhead incurred by JSGuard. The version with JSGuard is Firefox with our entire JSGuard system enabled.

6 Related Work

Detecting shellcode in JS objects is essential to protect vulnerable applications from JS based shellcode injection attacks. As §2.1 noted, existing shellcode detection approaches fall into 2 categories: *content analysis* and *hijack prevention*.

Content analysis is particularly popular in detecting shellcode from network messages. In [52], Toth and Kruegel proposed identifying exploit code by detecting NOP sleds. However, attacks can bypass this detection technique by either *excluding* NOP sleds or by using polymorphic techniques [11, 16, 30]. Chritodorescu and colleagues [12, 13] proposed techniques to detect malicious patterns in executables using semantic heuristics. Lakhotia and Eric in [27] used content analysis techniques to detect obfuscated calls in binaries. Chinchani and van den Berg proposed a rule-based scheme in [11]. Wang et al. proposed SigFree [55] that checks if network packets contain malicious codes using “push and call” patterns and the number of useful instructions in the longest possible execution chain. These methods are based on static analysis. Although they are efficient in detecting shellcode, they still can be thwarted by using binary obfuscation [5]. To improve detection completeness, Polychronakis et al. proposed a new network-level emulation approach [37, 38] to detect polymorphic shellcode. Gene [39] used network-level emulation with specific memory access pattern heuristics to detect shellcode for MS-Windows systems. Gu et al. proposed the virtual memory snapshot based emulation approach in end systems to detect shellcode in network messages before they are processed by network server programs [23]. ShellOS provides a framework leveraging hardware visualization to detect shellcode [46]. It requires users to dump the entire target process's states and load them into ShellOS in order to construct an emulation environment. A powerful shellcode analyzer named ”Shellzer” is proposed in [56]. It conducts analysis by instrumenting each instruction, which may incur undesirable overhead for online detection.

All these approaches are useful for detecting shellcode in network messages, but they are not directly applicable to detecting shellcode in JS strings, as such shellcode is not transmitted in its binary form. Instead, each byte of the shellcode is transmitted using its ASCII representation. In general, ASCII character sequences cannot be successfully decoded into the corresponding shellcode instruction sequences [18], though this is sometimes possible [35]. Nozzle is a well-known JS shellcode attack detection tool. It scans a heap object, interprets the object content to build a control flow graph (CFG), and then uses the CFG to check weather the content contains shellcode [43]. Egele et al. propose an approach that uses `libemu` [28] to check if the content of a JS string contains a sufficiently long valid instruction sequence using network-level emulation and GetPC code based heuristics. Hijack prevention based approaches can be used before or during shellcode execution. Such approaches include randomization [4, 6, 7, 26, 36], OS extension [3, 25] and flow tracking techniques [34, 42]. In general, these approaches have good detection completeness due to their extensive use of context information. However, their troubleshooting to find out the root cause is inefficient [55], which often requires heavy playback or log analysis. Recently, Gadaleta et al. proposed Bubble [22], a lightweight approach that encumbers complete execution of injected shellcode.

Recently, several machine learning based systems were proposed to detect malicious JS code. Zozzle applies Bayesian classification to hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that strongly predict malware [15]. Jsand [14] emulates JS code in a virtual browser environment using machine learning methods to capture malicious features. Prophiler [9] constructs a filter that can quickly discard benign pages and forward potentially malicious pages to heavyweight analysis tools. JSGuard can complement these systems by providing malicious code training samples.

We note that some works like Cujo [44] and Blade [29] can also prevent drive-by-download attacks. However, their focus differs from ours, which is malicious shellcode detection in JS code. These works cannot prevent in-memory execution of injected shellcode. We are aware that tools like [10, 19] have been proposed to audit JS activities, but they are not malicious shellcode detection systems.

7 Conclusion

In this paper, we have proposed a new methodology to detect JS shellcode that fully uses JS code execution environment information in an efficient manner. Following the methodology, we implemented JSGuard, a prototype malicious JS code detection system on Debian Linux. Extensive experiments with real traces and thousands of malicious shellcodes illustrate our detection system's performance with acceptable overhead and very few false negatives or false positives, which validated our methodology's promise for this purpose.

References

1. Alexa Top Sites. `http://www.alexa.com/topsites`.
2. P. Bania. TAPiON, 2005. `http://pb.specialised.info/all/tapion/`.
3. A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *USENIX Annual Technical Conf.*, 2000.
4. E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *CCS*, 2003.

5. U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic Analysis of Malicious Code. *Journal of Computer Virology*, 2006.
6. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security*, 2003.
7. S. Bhatkar and R. Sekar. Data Space Randomization. In *DIMVA*, 2008.
8. A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In *CHI*, 2000.
9. D. Canali, M. Cova, C. Kruegel, and G. Vigna. Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. In *WWW*, March 2011.
10. S. Chenette. Toorconx the ultimate deobfuscator, 2008. `http://www.toorcon.org/tcx/26_Chenette.pdf`.
11. R. Chinchani and E. van den Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *RAID*, 2005.
12. M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *USENIX Security*, 2003.
13. M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *IEEE S&P*, 2005.
14. M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *WWW*, 2010.
15. C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security*, 2011.
16. T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. van Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack*, 2003. `http://www.phrack.org`.
17. Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *ACSAC*, 2010.
18. M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending Browsers against Drive-by Downloads: Mitigating Heap-spraying Code Injection Attacks. In *DIMVA*, June 2009.
19. B. Feinstein and D. Peck. Caffeine Monkey. `http://www.secureworks.com/research/blog/wp-content/uploads/CaffeineMonkey_DEFCON15.pdf`.
20. S. Fogie, J. Grossman, R. Hansen, and A. Rager. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, May 2007.
21. S. Frei, T. Duebendorfer, G. Ollmann, and M. May. Understanding the web browser threat, 2008. In *DefCon 16 2008*, August 2008.
22. F. Gadaleta, Y. Younan, and W. Joosen. BuBBLe: A JavaScript Engine Level Countermeasure Against Heap-Spraying Attacks. In *ESSoS*, February 2010.
23. B. Gu, X. Bai, Z. Yang, A. C. Champion, and D. Xuan. Malicious Shellcode Detection with Virtual Memory Snapshots. In *INFOCOM*, pages 974–982, 2010.
24. C. Ionescu. GetPC code. `http://securityfocus.com/archive/82/327348/2006-01-03/1`.
25. G. S. Kc and A. D. Keromytis. e-nexsh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. In *ACSAC*, 2005.
26. G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *CCS*, 2003.
27. A. Lakhotia and U. Eric. Stack Shape Analysis to Detect Obfuscated Calls in Binaries. In *IEEE Int'l. Conf. on Source Code Analysis and Manipulation*, 2004.
28. libemu. `http://libemu.carnivore.it/`.
29. L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *CCS*, 2010.

30. S. Macaulay. ADMMutate: Polymorphic Shellcode Engine. `http://www.ktwo.ca/security.html`.
31. J. Mason, S. Small, F. Monrose, and G. MacManus. English Shellcode. In *CCS*, 2009.
32. D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting Execution Contexts for the Detection of Anomalous System Calls. In *RAID*, 2007.
33. F. F.-H. Nah. A Study on Tolerable Waiting Time: How Long are Web Users Willing to Wait? *Behaviour & IT*, 23(3):153–163, 2004.
34. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.
35. Obscou. Building IA32 'Unicode-Proof' Shellcodes. Phrack, 2003. `http://www.phrack.org/`.
36. PaX. `http://pax.grsecurity.net/docs/aslr.txt`.
37. M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level Polymorphic Shellcode Detection Using Emulation. In *DIMVA*, 2006.
38. M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-Based Detection of Non-Self-Contained Polymorphic Shellcode. In *RAID*, 2007.
39. M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *ACSAC*, Dec. 2010.
40. N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security*, 2008.
41. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In the Browser: Analysis of Web-based Malware. In *HotBots*, 2007.
42. F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO*, 2006.
43. P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *USENIX Security*, 2009.
44. K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *ACSAC*, December 2010.
45. Secunia. Secunia PSI study: 28% of all detected applications are insecure, 2007. `http://secunia.com/blog/11`.
46. K. Z. Snow, S. Krishnan, and F. Monrose. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security*, 2011.
47. A. Sotirov. Heap Feng Shui in JavaScript. In *BlackHat Europe*, 2007.
48. A. Sotirov and M. Dowd. Bypassing Browser Memory Protections. In *BlackHat*, 2008.
49. SpiderMonkey JavaScript engine. `http://www.mozilla.org/js/spidermonkey/`.
50. The Bastard Disassembly Environment. `http://bastard.sourceforge.net`.
51. The Metasploit Project. `http://www.metasploit.com`.
52. T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *RAID*, 2002.
53. Vulnerability Note VU#492515: Microsoft Internet Explorer HTML object memory corruption vulnerability. `http://www.kb.cert.org/vuls/id/492515`.
54. X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. In *ACSAC*, 2008.
55. X. Wang, C.-C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-Free Buffer Overflow Attack Blocker. In *USENIX Security*, 2006.
56. Y. Fratantonio, C. Kruegel, and G. Vigna. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *RAID*, 2011.