ShellOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks

Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose
Department of Computer Science
University of North Carolina at Chapel Hill,
{*kzsnow, krishnan, fabian*}*@cs.unc.edu*

Niels Provos
Google,
niels@google.com

Abstract

The availability of off-the-shelf exploitation toolkits for compromising hosts, coupled with the rapid rate of exploit discovery and disclosure, has made exploit or vulnerability-based detection far less effective than it once was. For instance, the increasing use of metamorphic and polymorphic techniques to deploy code injection attacks continues to confound signature-based detection techniques. The key to detecting these attacks lies in the ability to discover the presence of the injected code (or, shellcode). One promising technique for doing so is to examine data (be that from network streams or buffers of a process) and efficiently execute its content to find what lurks within. Unfortunately, current approaches for achieving this goal are not robust to evasion or scalable, primarily because of their reliance on software-based CPU emulators. In this paper, we argue that the use of software-based emulation techniques are not necessary, and instead propose a new framework that leverages hardware virtualization to better enable the detection of code injection attacks. We also report on our experience using this framework to analyze a corpus of malicious Portable Document Format (PDF) files and network-based attacks.

1 Introduction

In recent years, code-injection attacks have become a widely popular modus operandi for performing malicious actions on network services (e.g., web servers and file servers) and client-based programs (e.g., browsers and document viewers). These attacks are used to deliver and run arbitrary code (coined *shellcode*) on victims' machines, often enabling unauthorized access and control of the machine. In traditional code-injection attacks, the code is delivered by the attacker directly, rather than already existing within the vulnerable application, as in *return-to-libc* attacks. Depending on the specifics of the vulnerability that the attacker is targeting, injected code can take several forms, including source code for an interpreted scripting-language, intermediate byte-code, or natively-executable machine code [17].

Typically, though not always, the vulnerabilities exploited arise from the failure to properly define and reject improper input. These failures have been exploited by several classes of code-injection techniques, including buffer overflows [24], heap spray attacks [7, 36], and return oriented programming (ROP)-based attacks [3]. One prominent and contemporary example embodying these attacks involves the use of popular, cross-platform document formats, such as the Portable Document Format (PDF), to help compromise systems [37].

Malicious PDF files started appearing on the Internet a few years ago, and their rise steadily increased around the same time that Adobe Systems published their PDF format specifications [34]. Irrespective of when they first appeared, the reason for their rise in popularity as a method for compromising hosts is obvious: PDF is supported on all major operating systems, it supports a bewildering array of functionality (e.g., Javascript and Flash), and some applications (e.g., email clients) render them automatically. Moreover, the "stream objects" in PDF allow many types of encodings (or "filters" in the PDF language) to be used, including multi-level compression, obfuscation, and even encryption.

It is not surprising that malware authors quickly realized that these features can be used for nefarious purposes. Today, malicious PDFs are distributed via mass mailing, targeted email, and drive-by downloads [32]. These files carry an infectious payload that may come in the form of one or more embedded executables within the file itself[1], or contain shellcode that, after successful exploitation, downloads additional components.

The key to detecting these attacks lies in accurately discovering the presence of the shellcode in network payloads (for attacks on network services) or process buffers (for client-based program attacks). This, however, is a significant challenge because of the prevalent use of metamorphism (*i.e.,* the replacement of a set of instructions by a functionally-equivalent set of different instructions) and polymorphism (*i.e.,* a similar technique that hides a set of instructions by encoding—and later decoding—them), that allows the shellcode to change its appearance significantly from one attack to the next.

In this paper, we argue that a promising technique for detecting shellcode is to examine the input—be that network streams or buffers from a process—and efficiently *execute* its content to find what lurks within. While this idea is not new, we provide a novel approach based on a new kernel, called `ShellOS`, built specifically to address the shortcomings of current analysis techniques that use software-based CPU emulation to achieve the same goal (e.g., [6, 8, 13, 25, 26, 43]). Unlike these approaches, we take advantage of hardware virtualization to allow for far more efficient and accurate inspection of buffers by *directly executing* instruction sequences on the CPU. In doing so, we also reduce our exposure to evasive attacks that take advantage of discrepancies introduced by software emulation.

The remainder of the paper is organized as follows. We first present background information and related work in §2. Next, we discuss the challenges facing emulation-based approaches in §3. Our framework for supporting the detection and forensic analysis of code injection attacks is presented in §4. We provide a performance evaluation, as well as a case study of real-world attacks, in §5. Limitations of our current design are discussed in §6. Finally, we conclude in §7.

2 Background and Related Work

Early solutions to the problems facing signature-based detection systems attempted to find the presence of malicious code (for example, in network streams) by searching for tell-tale signs of executable code. For instance, Toth and Kruegel [38] applied a form of static analysis, coined *abstract payload execution*, to analyze the execution structure of network payloads. While promising, Fogla et al. [9] showed that polymorphism defeats this detection approach. Moreover, the underlying assumption that shellcode must conform to discernible structure on the wire was shown by several researchers [19, 29, 42] to be unfounded.

Going further, Polychronakis et al. [26] proposed the use of dynamic code analysis using emulation techniques to uncover shellcode in code injection attacks targeting network services. In their approach, the bytes off the wire from a network tap are translated into assembly instructions, and a simple software-based CPU emulator employing a read-decode-execute loop is used to execute the instruction sequences starting at each byte offset in the inspected input. The sequence of instructions starting from a given offset in the input is called an *execution chain*. The key observation is that to be successful, the shellcode must execute a valid execution chain, whereas instruction sequences from benign data are likely to contain invalid instructions, access invalid memory addresses, cause general protection faults, etc. In addition, valid malicious execution chains will exhibit one or more observable behaviors that differentiate them from valid benign execution chains. Hence, a network stream can be flagged as malicious if there is a single execution chain within the inspected input that does not cause fatal faults in the emulator before malicious behavior is observed. This general notion of *network-level emulation* has proven to be quite useful, and has garnered much attention of late (e.g., [13, 25, 41, 43]).

Recently, Cova et al. [6] and Egele et al. [8] extended this idea to protect web browsers from so-called "heap-spray" attacks, where an attacker coerces an application to allocate many objects containing malicious code in order to increase the success rate of an exploit that jumps to locations in the heap [36]. These attacks are particularly effective in browsers, where an attacker can use JavaScript to allocate many malicious objects [4, 35]. Heap spraying has been used in several high profile attacks on major browsers and document readers. Several Common Vulnerabilities and Exposure (CVE) disclosures have been released about these attacks in the wild. To the best of our knowledge, all the aforementioned exploit detection approaches employ software-based CPU emulators to detect shellcode in heap objects.

Finally, we note that although runtime analysis of payloads using software-based CPU emulation techniques has been successful in detecting exploits in the wild [8, 27], the use of software emulation makes them susceptible to multiple methods of evasion [18, 21, 33]. Moreover, as we show later, software emulation is not scalable. Our objective in this paper is to forgo software-based emulation altogether, and explore the design and implementation of components necessary for robust detection of code injection attacks.

3 Challenges for Software-based CPU Emulation Detection Approaches

As alluded to earlier, prior art in detecting code injection attacks has applied a simple read-decode-execute approach, whereby data is translated into its corresponding instructions, and then emulated in software. Obviously, the success of such approaches rests on accurate software emulation; however, the instruction set for modern CISC architectures is very complex, and so it is unlikely that software emulators will ever be bug free [18].

As a case-in-point, the popular and actively developed `QEMU` emulator [2], which employs more advanced emulation techniques based on dynamic binary translation, does not faithfully emulate the FPU-based Get Program Counter (GetPC) instructions, such as `fnstenv`[2]. Consequently, some of the most commonly used code injection attacks fail to execute properly, including those encoded with Metasploit's popular "shikata ga nai" encoder and three other encoders from its arsenal that rely on this GetPC instruction to decode their payload. While this may be a boon to `QEMU` users employing it for full-system virtualization (as one rarely requires a fully faithful `fnstenv` implementation for normal application usage), using this software emulator as-is for injected code detection would be fairly ineffective. In fact, we abandoned our earlier attempts at building a `QEMU`-based detection system for exactly this reason.

To address accurate emulation of machine instructions typically used in code injection attacks, special-purpose CPU emulators (e.g. `nemu` [28], `libemu` [1]) were developed. Unfortunately, they suffer from a different problem: large subsets of instructions rarely used by injected code are skipped when encountered in the instruction stream. The result is that any discrepancy between an emulated instruction and the behavior on real hardware potentially allows shellcode to evade detection by altering its behavior once emulation is detected [21, 33]. Indeed, the ability to detect emulated enviroments is already present in modern exploit toolkits.

Arguably, a more practical limitation of emulation-based detection is that of performance. When this approach is used in network-level emulation, for example, the overhead can be non-trivial since (*i*) the vast majority of network streams will contain benign data, some of which might be significant in size, (*ii*) successfully detecting even non-sophisticated shellcode can require the execution of thousands of instructions, and (*iii*) a separate execution chain must be attempted for each offset in a network stream because the starting location of injected code is unknown.

To avoid these obstacles, the current state of practice is to limit run-time analysis to the first n bytes (e.g., 64kb) of one side of a network stream, to examine flows to only known servers or from known services, or to terminate execution after some threshold of instructions (e.g., 2048) has been reached [25, 27, 43]. It goes without saying that imposing such stringent run-time restrictions inevitably leads to the possibility of missing attacks (e.g., in the unprocessed portions of streams).

One might argue that more advanced software-based emulation techniques such as dynamic binary translation [30] could offer significant performance enhancements over the simple emulation used in current state-of-the-art dynamic shellcode detectors. However, the performance benefit of dynamic binary translation hinges on the assumption that code blocks are translated once, but executed many times. While this assumption holds true with typical application usage, executing random streams of data (as in network-level emulation) results in short instruction sequences ending in a fault, rather than a structured program flow. Furthermore, dynamic binary translation still has the problem of emulation accuracy.

Lastly, it is common for software-based CPU emulation techniques to omit processing of some execution chains as a performance-boosting optimization (e.g., only executing instruction sequences that contain a GetPC instruction, or skipping an execution chain if the starting instruction was already executed during a previous execution chain). Unfortunately, such optimizations are unsafe, in that they are susceptible to evasion. For instance, in the former case, metamorphic code may evade detection by, for example, pushing data representing a GetPC instruction to the stack and then executing it.

```
—————————————— begin snippet ——————————————
0 exit:
1  in al, 0x7           ; Chain 1
2  mov eax, 0xFF        ; Chain 2 begins
3  mov ebx, 0x30        ; Chain 2
4  cmp eax, 0xFF        ; Chain 2
5  je exit              ; Chain 2 ends
6  mov eax, fs:[ebx]    ; Chain 3 begins
   ...
——————————————— end snippet ———————————————
```

Figure 1: Sample instruction sequence

In the latter case, consider the sequence shown in Figure 1. The first execution chain ends after a single privileged instruction. The second execution chain executes instructions 2 to 5 before ending due to a conditional jump to a privileged instruction. Now, since instructions 3, 4, and 5 were already executed in the second execution chain they are skipped (as a beginning offset) as a performance optimization. The third execution chain begins at instruction 6 with an access to the Thread Environment Block (TEB) data structure to the offset specified by `ebx`. Had the execution chain beginning at instruction 3 not been skipped, `ebx` would be loaded with `0x30`. Instead, `ebx` is now loaded with a random value set by the emulator at the beginning of each execution chain. Thus, if detecting an access to the memory location at `fs:[0x30]` is critical to detecting injected code, the attack will be missed.

4 Our Approach: ShellOS

Unlike prior approaches, we take advantage of the observation that the most widely used heuristics for shellcode detection exploit the fact that, to be successful, the injected shellcode typically needs to read from memory (e.g., from addresses where the payload has been mapped in memory, or from addresses in the Process Environment Block (PEB)), write the payload to some memory area (especially in the case of polymorphic shellcode), or transfer flow to newly created code [16, 22, 23, 25–28, 41, 43]. For instance, the execution of shellcode often results in the resolution of shared libraries (DLLs) through the PEB. Rather than tracing each instruction and checking whether its memory operands can be classified as "PEB reads," we allow instruction sequences to execute directly on the CPU using hardware virtualization, and only trace specific memory reads, writes, and executions through hardware-supported paging mechanisms.

Our design for enabling hardware-support of code injection attacks is built upon a virtualization solution [12] known as *Kernel-based Virtual Machine* (KVM). We use the KVM hypervisor to abstract Intel VT and AMD-V hardware virtualization support. At a high level, the KVM hypervisor is composed of a privileged domain and a virtual machine monitor (VMM). The privileged domain is used to provide device support to unprivileged guests. The VMM, on the other hand, manages the physical CPU and memory and provides the guest with a virtualized view of the system resources.

In a hardware virtualized platform, the VMM only mediates processor events (e.g., via instructions such as VMEntry and VMExit on the Intel platform) that would cause a change in the entire system state, such as physical device IO, modifying CPU control registers, etc. Therefore, it no longer emulates guest instruction executions as with software-based CPU emulation; execution happens directly on the processor, without an intermediary instruction translation. We take advantage of this design to build a new kernel, called ShellOS, that runs as a guest OS using KVM and whose sole task is to detect and analyze code injection attacks. The high-level architecture is depicted in Figure 2.

4.1 The SHELLOS Interface

ShellOS can be viewed as a black box, wherein a buffer is supplied to ShellOS by the privileged domain for inspection via an API call. ShellOS performs the analysis and reports (1) if injected code was found, (2) the location in the buffer where the shellcode was found, and (3) a log of the actions performed by the shellcode.

A library within the privileged domain provides the ShellOS API call, which handles the sequence of actions required to initialize guest mode via the KVM ioctl interface. One notable feature of initializing guest mode in KVM is the assignment of guest physical memory from a userspace-allocated buffer. We use this feature to satisfy a critical requirement — that is, efficiently moving buffers into ShellOS for analysis. Since offset zero of the userspace-allocated memory region corresponds to the guest physical address of 0x0, we can reserve a fixed memory range within the guest address space where the privileged domain library writes the buffers to be analyzed. These buffers are then directly accessible to the ShellOS guest at the predefined physical address.

The privileged domain library also optionally allows the user to specify a process snapshot for ShellOS to use as the default environment. The details about this snapshot are given later in §4.5, but for now it is sufficient to note that the intention is to allow the user to analyze buffers in an environment as similar as possible to what the injected code would expect. For example, a user analyzing buffers extracted from a PDF process may provide an Acrobat Reader snapshot, while one analyzing Flash objects might supply an Internet Explorer snapshot. While malicious code detection may typically occur without this extra data, it provides a realistic environment for our post facto diagnostics.

When the privileged domain first initializes ShellOS, it completes its boot sequence (detailed next) and issues a VMExit. When the ShellOS API is called to analyze a buffer, it is copied to the fixed shared region before a VMEnter is issued. ShellOS completes its analysis and writes the result to the shared region before issuing another VMExit, signaling that the kernel is ready for another buffer. Finally, we build a thread pool into the library where-in each buffer to be analyzed is added to a work queue and one of n workers dequeues the job and analyzes the buffer in a unique instance of ShellOS.

4.2 The SHELLOS Kernel

To set up our execution environment, we initialize the Global Descriptor Table (GDT) to mimic a Windows environment. More specifically, code and data entries are added for user and kernel modes using a flat 4GB memory model, a Task State Segment (TSS) entry is added that denies all usermode IO access, and a special entry that maps to the virtual address of the Thread Environment Block (TEB) is added. We set the auxiliary FS segment register to select the TEB entry, as done by the Windows kernel. Therefore, regardless of where the TEB is mapped into memory, code (albeit benign or malicious) can always access the data structure at FS:[0]. This "feature" is commonly used by injected code to find shared library locations, and indeed, access to this region of memory has been used as a heuristic for identifying injected code [28].

Virtual memory is implemented with paging, and mirrors that of a Windows process. Virtual addresses above

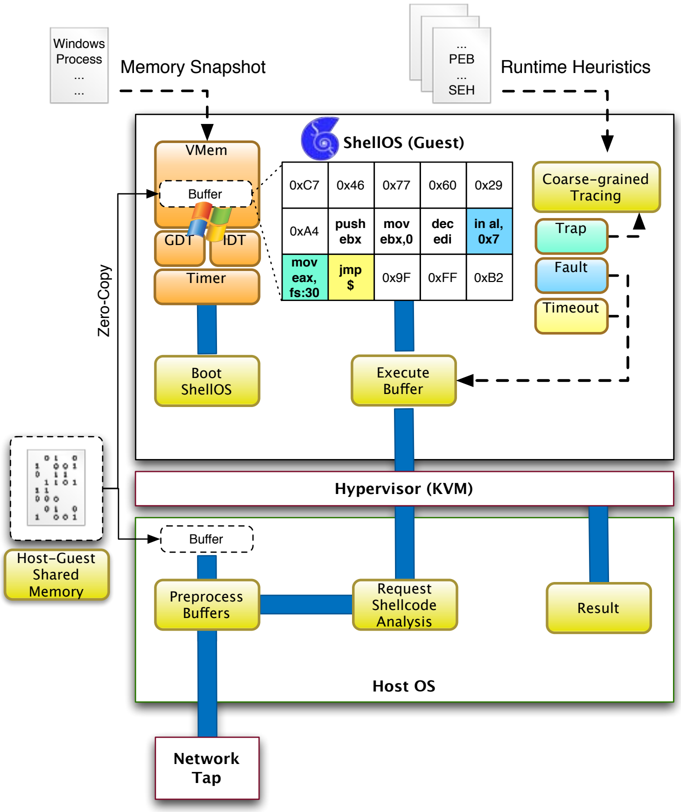

Figure 2: Architecture for detecting code injection attacks. The `ShellOS` platform includes the `ShellOS` operating system and host-side interface for providing buffers and extending `ShellOS` with custom memory snapshots and runtime detection heuristics. As shown, buffers are analyzed from reassembled `TCP` connections collected on a network tap; however `ShellOS` may be used as a component in any framework that requires analysis of injected code.

3GB are reserved for the `ShellOS` kernel. The kernel supports loading arbitrary snapshots created using the `minidump` format [20] (e.g., used in tools such as `WinDBG`). The `minidump` structure contains the necessary information to recreate the state of the running process at the time the snapshot was taken. Once all regions in the snapshot have been mapped, we adjust the TEB entry in the Global Descriptor Table to point to the actual TEB location in the snapshot.

Control Loop Recall that `ShellOS`' primary goal is to enable fast and accurate detection of input containing shellcode. To do so, we must support the ability to execute the instruction sequences starting at every offset in the inspected input. Execution from each offset is required since the first instruction of the shellcode is unknown. The control loop in `ShellOS` is responsible for this task. Once `ShellOS` is signaled to begin analysis, the `fpu,mmx`, `xmm`, and general purpose registers are randomized to thwart injection attacks that try to hinder analysis by guessing fixed register values (set by `ShellOS`) and end execution early upon detection of these conditions. The program counter is set to the address of the buffer being analyzed. Buffer execution begins when `ShellOS` transitions to usermode with the `iret` instruction. At this point, instructions are executed directly on the CPU in usermode until execution is interrupted by a *fault*, *trap*, or *timeout*. The control loop is therefore completely interrupt driven.

We define a fault as an unrecoverable error in the instruction stream, such as attempting to execute a privileged instruction (e.g., the `in al, 0x7` instruction in Figure 2), or encountering an invalid opcode. The kernel is notified of a fault through one of 32 interrupt vectors indicating a processor exception. The Interrupt Descriptor Table (IDT) points all fault-generating interrupts to a generic assembly-level routine that resets usermode state before attempting the next execution chain.[3]

We define a trap, on the other hand, as a recoverable exception in the instruction stream (e.g., a page fault resulting from a needed, but not yet paged-in, virtual address), and once handled appropriately, the instruction stream continues execution. Traps provide an opportunity to coarsely trace some actions of the executing code, such as reading an entry in the TEB. To deal with instruction sequences that result in infinite loops, we currently use a rudimentary approach wherein ShellOS instructs the programmable interval timer (PIT) to generate an interrupt at a fixed frequency. When this timer fires twice in the current execution chain (guaranteeing at least 1 tick interval of execution time), the chain is aborted. Since the PIT is not directly accessible in guest mode, KVM emulates the PIT timer via privileged domain timer events implemented with hrtimer, which in turn uses the High Precision Event Timer (HPET) device as the underlying hardware timer. This level of indirection imposes an unavoidable performance penalty because external interrupts (e.g. ticks from a timer) cause a VMExit.

Furthermore, the guest must signal that each interrupt has been handled via an End-of-Interrupt (EOI). The problem here is that EOI is implemented as a physical device IO instruction which requires a second VMExit for each tick. The obvious trade-off is that while a higher frequency timer would allow us to exit infinite loops quickly, it also increases the overhead associated with entering and exiting guest mode (due to the increased number of VMExits). To alleviate some of this overhead, we place the KVM-emulated PIT in what is known as Auto-EOI mode. This mode allows new timeout interrupts to be received without requiring a device IO instruction to acknowledge the previous interrupt. In this way, we effectively cut the overhead in half. We return later to a discussion on setting appropriate timer frequencies, and its implications for run-time performance.

The complete ShellOS kernel is composed of 2471 custom lines of C and assembly code.

4.3 Detection

The ShellOS kernel provides an efficient means to execute arbitrary buffers of code or data, but we also need a mechanism for determining if these execution sequences represent injected code. One of our primary contributions in this paper is the ability to modularly use existing runtime heuristics in an efficient and accurate framework that does not require tracing every machine-level instruction, or performing unsafe optimizations. A key insight towards this goal is the observation that existing reliable detection heuristics really do not require fine-grained instruction-level tracing, rather, coarsely tracing memory accesses to specific locations is sufficient.

Towards this goal, a handful of approaches are readily available for efficiently tracing memory accesses; e.g., using hardware supported debug registers, or exploring virtual memory based techniques. Hardware debug registers are limited in that only a few memory locations may be traced at one time. Our approach, based on virtual memory, is similar in implementation to stealth breakpoints [40] and allows for an unlimited number of memory traps to be set to support multiple runtime heuristics defined by an analyst.

Recall that an instruction stream will be interrupted with a *trap* upon accessing a memory location that generates a *page fault*. We may therefore force a trap to occur on access to an arbitrary virtual address by clearing the present bit of the page entry mapping for that address. For each address that requires tracing we clear the corresponding present bit and set the OS reserved field to indicate that the kernel should trace accesses to this entry. When a page fault occurs, the interrupt descriptor table (IDT) directs execution to an interrupt handler that checks these fields. If the OS reserved field indicates tracing is not requested, then the page fault is handled according to the region mappings defined in the process' snapshot. Regardless of where the analyzed buffers originate from (e.g., a network packet or a heap object) a Windows process snapshot is always loaded in ShellOS in order to populate OS data structures (e.g., the TEB), and to load data commonly present (e.g., shared libraries) when injected code executes.

When a page entry does indicate that tracing should occur, and the faulting address (accessible via the CR2 register) is in a list of desired address traps (provided, for example, by an analyst), the page fault must be logged and appropriately handled. In handling a page fault resulting from a trap, we must first allow the page to be accessed by the usermode code, then reset the trap immediately to ensure trapping future accesses to that page. To achieve this, the handler sets the present bit in the page entry (enabling access to the page) and the TRAP bit in the flags register, then returns to the usermode instruction stream. As a result, the instruction that originally caused the page fault is now successfully executed before the TRAP bit forces an interrupt. The IDT then forwards the interrupt to another handler that unsets the TRAP and present bits so that the next access to that location can be traced. Our approach allows for tracing of any virtual address access (read,write, execute), without a predefined limit on the number of addresses to trap.

Detection Heuristics ShellOS, by design, is not tied to any specific set of behavioral heuristics. Any heuristic based on memory reads, writes, or executions can be supported with coarse-grained tracing. To highlight the strengths of ShellOS, we chose to implement the PEB heuristic proposed by Polychronakis et al. [28]. That particular heuristic was chosen for its simplicity, as well as the fact that it has already been shown to be successful in detecting a wide array of Windows shellcode. This heuristic detects injected code that parses the process-level TEB and PEB data structures in order to locate the base address of shared libraries loaded in memory. The TEB contains a pointer to the PEB (address `FS:[0x30]`), which contains a pointer to yet another data structure (*i.e.*, LDR_DATA) containing several linked lists of shared library information.

The detection approach given in [28] checks if accesses are being made to the PEB pointer, the LDR_DATA pointer, and any of the linked lists. To implement their detection approach, we simply set a trap on each of these addresses and report that injected code has been found when the necessary conditions are met. This heuristic fails to detect certain cases, but we reiterate that any number of other heuristics could be chosen instead. We leave this as future work.

4.4 Diagnostics

Although efficient and reliable identification of code injection attacks is an important contribution of this paper, the forensic analysis of the higher-level actions of these attacks is also of significant value to security professionals. To this end, we provide a method for reporting forensic information about a buffer where shellcode has been detected. Again, we take advantage of the memory snapshot facility discussed earlier (§ 4.5) to obtain a list of virtual addresses associated with API calls for various shared libraries. We place traps on these addresses, and when triggered, a handler for the corresponding call is invoked. That handler pops function parameters off the usermode stack, logs the call and its supplied parameters, performs actions needed for the successful completion of that call (e.g., allocating heap space), and then returns to the injected code.

Obviously, due to the myriad of API calls available, one cannot expect the diagnostics to be complete. Keep in mind, however, that the lack of completeness in our diagnostics facility is independent of the actual detection of injected code. The ability to extend the level of diagnostic information is straightforward, but tedious. That said, as shown later, we are able to provide a wealth of diagnostic information on a diverse collection of self-contained [27] shellcode injection attacks.

4.5 Extensibility

The capabilities provided by `ShellOS` are but one component in an overall framework necessary to detect code injection attacks. This larger framework should support the loading of custom process snapshots and arbitrary shellcode detection heuristics, each defined by a list of read, write, or execute memory traps. Since `ShellOS` only detects and diagnoses the buffers of data provided, there must be some mechanism for providing buffers of data we suspect contain injected code. To this end, we built two platforms that rely on `ShellOS` to scan buffers for injected code; one to detect client-based program attacks such as the malicious PDFs discussed earlier, and another to detect attacks on network services that operates as a network intrusion detection system.

Supporting Detection of Code Injection in Client-based Programs: To showcase `ShellOS`' promise as a platform upon which other modules can be built, we implemented a lightweight memory monitoring facility that allows `ShellOS` to scan buffers created by documents loaded in the process space of a prescribed reader application. In this context, a document is any file or object that may be opened with it's corresponding program, such as a PDF, Microsoft Word document, Flash object, HTML page, etc. This platform may be useful to an enterprise as a network service wherein documents are automatically sent for analysis (e.g. by extraction from network streams or an email server) or manually submitted by an analyst in a forensic investigation.

The approach we take to detect shellcode in malicious documents is to let the reader application handle rendering of the content while monitoring any buffers created by it, and signaling `ShellOS` to scan these buffers for shellcode (using existing heuristics). This approach has several advantages. An important one is that we do not need to worry about recreating any document object model, handling obfuscated javascript, or dealing with all the other idiosyncrasies that pose challenges for other approaches [6, 8, 39]. We simply need to analyze the buffers created when rendering the document in a quarantined environment. The challenge lies in doing all of this as efficiently as possible.

To support this goal, we provide a monitoring facility that is able to snapshot the memory contents of processes. The snapshots are constructed in a manner that captures the entire process state, the virtual memory layout, as well as all the code and data pages within the process. The data pages contain the buffers allocated on the heap, while the code pages contain all the system modules that must be loaded by `ShellOS` to enable analysis. Our memory tracing facility includes less than 900 lines of custom C/C++ code. A high level view of the approach is shown in Figure 3.

This functionality was built specifically for the Windows OS and can support any application running on Windows. The memory snapshots are created using custom software that attaches to an arbitrary application process and stores contents of memory using the functionality provided by Windows' debug library (DbgHelp). We capture buffers that are allocated on the heap (*i.e.*, pages mapped as `RW`), as well as thread and module information. The results are stored in `minidump` format,

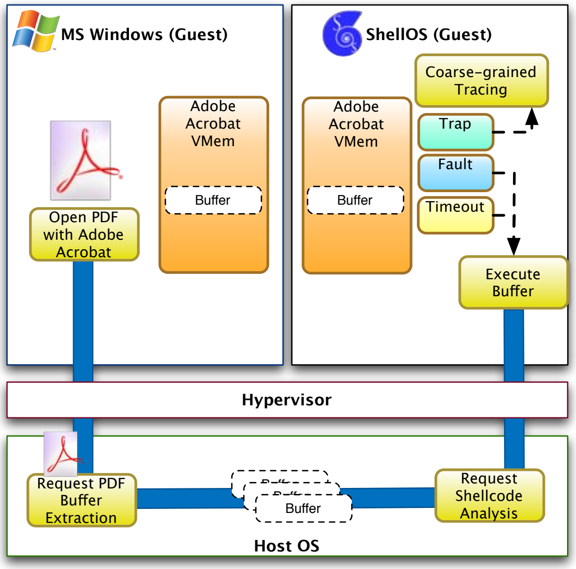

Figure 3: A platform for analyzing process buffers using ShellOS

which contains all the information required to recreate the process within ShellOS, including all dlls, the PEB/TEB, register state, the heap and stack, and the virtual memory layout of these components.

Supporting Detection of Code Injection in Network Services: Another use-case for ShellOS is detecting code injection attacks targeting network services. While the shellcode embedded in client-based program code injection attacks is typically obfuscated in multiple layers of encoding (e.g. compressed form → javascript → shellcode), attacks on network services are often present directly as executable shellcode on the wire. As noted by Polychronakis et al. [26], we may use this observation to build a platform to detect code injection attacks on network services by reassembling observed network streams and executing each of these streams. This platform may be used in an enterprise as a component of an network intrusion detection system or for post-facto analysis of a network capture in a forensic investigation.

5 Evaluation

In the analysis that follows, we first examine ShellOS' ability to faithfully execute network payloads and successfully trigger the detection heuristics when shellcode is found. Next, we examine the performance benefits of the ShellOS framework when compared to software-emulation. We also report on our experience using ShellOS to analyze a collection of suspicious PDF documents. All experiments were conducted on an Intel Xeon Quad Processor machine with 32 GB of memory. The host OS was Ubuntu with kernel version 2.6.35.

Encoder	Nemu	ShellOS
countdown	Y	Y
fnstenv_mov	Y	Y
jmp_call_additive	Y	Y
shikata_ga_nai	Y	Y
call4_dword_xor	Y	Y
alpha_mixed	Y	Y
alpha_upper	N	Y
TAPiON	Y*	Y

Table 1: Off-the-Shelf Shellcode Detection.

5.1 Performance

To evaluate our performance, we used Metasploit to launch attacks in a virtualized environment. For each encoder, we generated 100s of attack instances by randomly selecting 1 of 7 exploits, 1 of 9 self-contained payloads that utilize the PEB for shared library resolution, and randomly generated parameter values associated with each type of payload (e.g. download URL, bind port, etc.). As the attacks launched, we captured the network traffic for later network-level buffer analysis.

We also encoded several payload instances using an advanced polymorphic engine, called TAPiON[4]. TAPiON incorporates features designed to thwart emulation. Each of the encoders we used (see Table 1) are considered to be *self-contained* [25] in that they do not require additional contextual information about the process they are injected into in order to function properly. Indeed, we do not specifically address non-self-contained shellcode in this paper.

For the sake of comparison, we chose a software-based solution (called Nemu [28]), that is reflective of the current state of the art. Nemu and ShellOS both performed well in detecting all the instances of the code injection attacks developed using Metasploit, with a few exceptions.

Surprisingly, Nemu failed to detect shellcode generated using the alpha_upper encoder. Since the encoder payload relies on accessing the PEB for shared library resolution, we expected both Nemu and ShellOS to trigger this detection heuristic. We speculate that Nemu is unable to handle this particular case because of inaccurate emulation of its particular instruction sequences—underscoring the need to directly execute the shellcode on hardware.

More pertinent to the discussion is that while the software-based emulation approach is capable of detecting shellcode generated with the TAPiON engine, performance optimization limits its ability to do so. The TAPiON engine attempts to confound detection by basing its decoding routines on timing components (namely, the RDTSC instruction) and uses a plethora of CPU-intensive coprocessor instructions in long loops to slow runtime-analysis. These long loops quickly reach Nemu's default execution threshold (2048) prior to any heuristic being triggered. This is particularly problematic because no GetPC instruction is executed until these loops complete.

Furthermore, software-based emulators simply treat the majority of coprocessor instructions as NOPs. While TAPiON does not currently use the result of these instructions in its decoding routine, it only takes minor changes to the out-of-the-box engine to incorporate these results and thwart detection (hence the "*" in Table 1). ShellOS, on the other hand, fully supports all coprocessor instructions with its direct CPU execution.

More problematic for these classes of approaches is that successfully detecting code encoded by engines such as TAPiON can require following very long execution chains (e.g., well over $60,000$ instructions). To examine the runtime performance of our prototype, we randomly generated 1000 benign inputs, and set the instructions thresholds (in *both* approaches) to the levels required to detect instances of TAPiON shellcode.

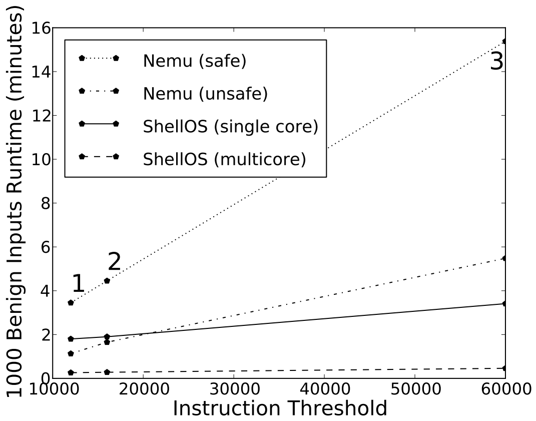

Figure 4: ShellOS Performance

Since ShellOS currently cannot directly set an instruction threshold (due to the coarse-grained tracing approach), we approximate the required threshold by adjusting the execution chain timeout frequency. As the timer frequency increases, the number of instructions executed per execution chain decreases. Thus, we experimentally determined the maximum frequency needed to execute the TAPiON shellcodes that required $10k$, $16k$, and $60k$ instruction executions to complete their loops. These timer frequencies are 5000HZ, 4000HZ, and 1000HZ, respectively. Note that in the common case, ShellOS can execute many more instructions, depending on the speed of individual instructions. TAPiON code, however, is specifically designed to use the slower FPU-based instructions. (ShellOS can execute over 4 million fast NOP instructions in the same time interval that only 60k FPU-heavy instructions are executed.)

The results are shown in Figure 4. The labeled points on the lineplot indicate the minimum execution chain length required to detect the three representative TAPiON samples. For completeness, we show the performance of Nemu with and without unsafe execution chain pruning (see §3). When unsafe pruning is used, software-emulation does better than ShellOS on a single core at very low execution thresholds. This is not too surprising, as the higher clock frequencies required to support short execution chains in ShellOS incur additional overhead (see §4). However, with longer execution chains, the real benefit of ShellOS becomes apparent—ShellOS (on a single core) is an order of magnitude faster than Nemu when unsafe execution chain pruning is disabled. Finally, we observe that the worker queue provided by the ShellOS host-side library efficiently multi-processes buffer analysis, and demonstrates that multi-processing offers a viable alternative to the unsafe elimination of execution chains.

A note on 64-bit architectures The performance of ShellOS is even more compelling when one takes into consideration the fact that in 64-bit architectures, program counter relative addressing is allowed—hence, there is no need for shellcode to use any form of "Get Program Counter" code to locate its address on the stack; a limitation that has been widely used to detect traditional 32-bit shellcode using (very) low execution thresholds. This means that as 64-bit architectures become commonplace, shellcode detection approaches using dynamic analysis must resort to heuristics that require the shellcode to fully decode. The implications are that the requirement to process long execution chains, such as those already exhibited by today's advanced engines (e.g., Hydra [29] and TAPiON), will be of far more significance than it is today.

5.2 Throughput

To better study our throughput on network streams, we built a testbed consisting of 32 machines running FreeBSD 6.0 and generated traffic using a state-of-the-art traffic generator, *Tmix* [15]. The network traffic is routed between the machines using Linux-based software routers. The link between the two routers is tapped using a gigabit fiber tap, with the traffic diverted to our detection appliance (i.e., running ShellOS or Nemu), as well as to a network monitor that constantly monitors the network for throughput and losses. The experimental setup is shown in Figure 5.

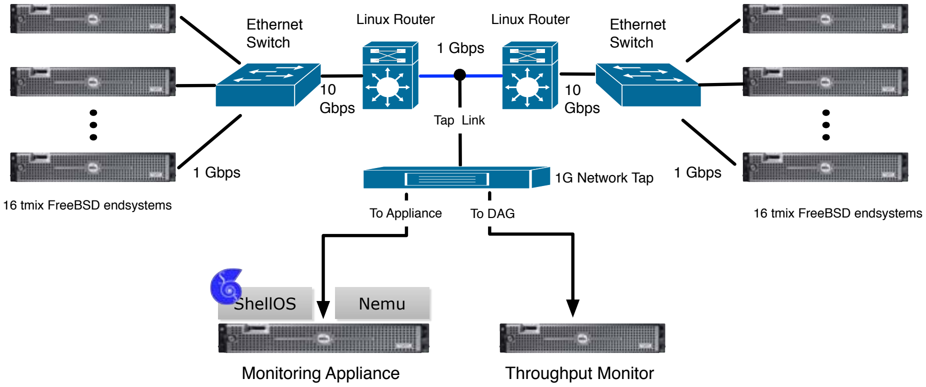

Figure 5: Experimental testbed with end systems generating traffic using Tmix. Using a network tap, we monitor the throughput on one system, while `ShellOS` or `Nemu` attempt to analyze all traffic on another system.

Tmix synthetically regenerates TCP traffic that matches the statistical properties of traffic observed in a given network trace; this includes source level properties such as file and object size distributions, number of simultaneously active connections and also network level properties such as round trip time. Tmix also provides a block resampling algorithm to achieve a target throughput while preserving the statistical properties of the original network trace.

We supply Tmix with a network trace of HTTP connections captured on the border links of UNC-Chapel Hill in October, 2009[5]. The trace represents 1-hour of activity, which is more than long enough to capture distributions for many statistical measures indistinguishable from longer traces [14]. Using Tmix block resampling, we run two 1-hour experiments based on the original trace where Tmix attempts to maintain a throughput of 100Mbps in the first experiment and 350Mbps in the second experiment. The actual throughput fluctuates some as Tmix maintains statistical properties observed in the original network trace. We repeat each experiment with the same seed (to generate the same traffic) using both `Nemu` and `ShellOS`.

Both `ShellOS` and `Nemu` are configured to only analyze traffic from the connection initiator, as we are targeting code injection attacks on network services. We analyze up to one megabyte of a network connection (from the initiator) and set an execution threshold of $60k$ instructions (see section §5.1). Neither `ShellOS` or `Nemu` perform any instruction chain pruning (e.g. we try execution from every position in every buffer) and use only a single cpu core.

Figure 6 shows the results of the network experiments. The bottom subplot shows the traffic throughput generated over the course of both 1-hour experiments. The 100Mbps experiment actually fluctuates from 100-160Mbps, while the 350Mbps experiment nearly reaches 500Mbps at some points. The top subplot depicts the number of buffers analyzed over time for both `ShellOS` and `Nemu` with both experiments. Note that one buffer is analyzed for each connection containing data from the connection initiator. The plot shows that the maximum number of buffers per second for `Nemu` hovers around 75 for both the 100Mbps and 350Mbps experiments with significant packet loss observed in the middle subplot. `ShellOS` is able to process around 250 buffers per second in the 100Mbps experiment with zero packet loss and around 750 buffers per second in the 350Mbps experiment with intermittent packet loss. That is, `ShellOS` is able to process all buffers with 1 CPU core, without loss, on a network with sustained 100Mbps network throughput, while `ShellOS` is on the cusp of its maximum throughput on 1 CPU core on a network with sustained 350Mbps network throughput (and spikes up to 500Mbps). In these tests, we received no false positives for either `ShellOS` or `Nemu`.

Our experimental network setup, unfortunately, is not currently able to generate sustained throughput greater than the 350Mbps experiment. Therefore, to demonstrate `ShellOS`' scalability in leveraging multiple CPU cores, we instead turn to an analysis of the `libnids` packet queue size in the 350Mbps experiment. We fix the maximum packet queue size at 100k, then run the 350Mbps experiment 4 times utilizing 1, 2, 4, and 14 cores. When the packet queue size reaches the maximum, packet loss occurs. The average queue size should be as low as possible to minimize the chance of packet loss due to sudden spikes in network traffic, as observed in the middle subplot of Figure 6 for the 350Mbps `ShellOS` experiment. Figure 7 shows the CDF of the average packet queue size over the course of each 1-hour experiment run with a different number of CPU cores. The figure shows

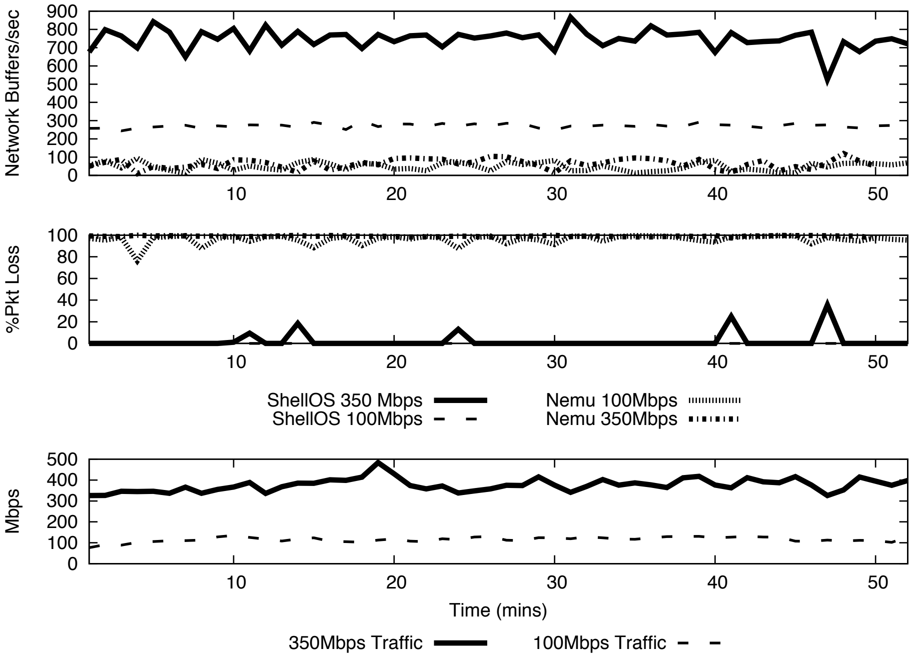

Figure 6: `ShellOS` network throughput performance.

that using 2 cores reduces the average queue size by an order of magnitude, 4 cores reduces average queue size to less than 10 packets, and 14 cores is clearly more than sufficient for 350Mbps sustained network traffic. This evidence suggests that multi-core `ShellOS` may be capable of monitoring links with much greater throughput than we were able to generate in our experiments.

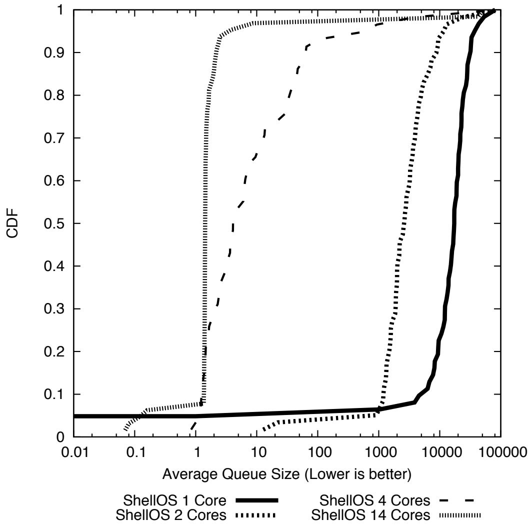

Figure 7: CDF of the average packet queue size as the number of `ShellOS` CPU cores is scaled.

5.3 Case Study: PDF Code Injection

We now report on our experience using this framework to analyze a collection of 427 malicious PDFs. These PDFs were randomly selected from a larger subset of suspicious files flagged by a large-scale web malware detection system. Each PDF is labeled with a Common Vulnerability Exposure (CVE) number (or "Unknown" tag). Of these files, 22 were corrupted, leaving us with a total of 405 files for analysis. We also use a collection of 179 benign PDFs from various USENIX conferences.

We launch each document with Adobe Reader and attach the memory facility to that process. We then snapshot the heap as the document is rendered, and wait until the heap buffers stop growing. 374 of the 405 malicious PDFs resulted in a unique set of buffers. `ShellOS` is then signaled that the buffers are ready for inspection. Note that we only generate the process layout once per application (e.g., Reader), and subsequent snapshots only contain the heap buffers.

Figure 8 shows the size distribution of heap buffers extracted from benign and malicious PDFs. Notice that $\approx 60\%$ of the buffers extracted from malicious PDF are 512K long. This striking feature can be attributed to the heap allocation strategy used by the Windows OS, whereby chunks of 512K and higher are memory aligned at 64K boundaries. As noted by Ding et al. [7], attackers can take advantage of this alignment to increase the success rate of their attacks (e.g., by providing a more

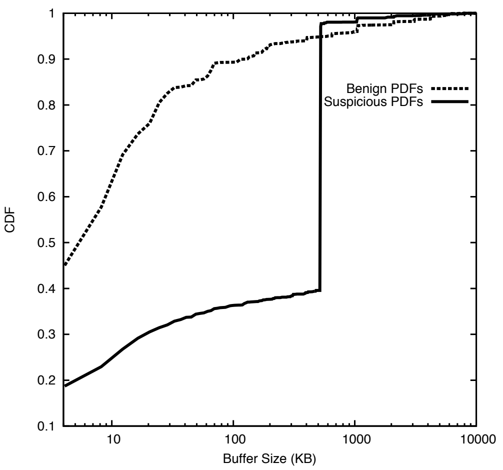

Figure 8: CDF of sizes of the extracted buffers

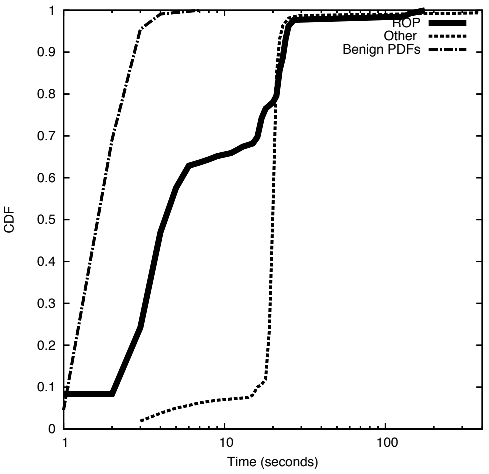

Figure 9: Elapsed time for extracting heap objects

predictable landing spot for the shellcode when used in conjunction with large NOP-sleds).

CVE	Detected
CVE-2007-5659	2
CVE-2008-2992	10
CVE-2009-4324	12
CVE-2009-2994	1
CVE-2009-0927	33
CVE-2010-0188	53
CVE-2010-2883	70
Unknown	144

Table 2: CVE Distribution for Detected Attacks

Table 2 provides a breakdown of the corresponding CVE listings for the 325 unique code injection attacks we detected. Interestingly, we were able to detect 70 attacks using Return Oriented Programming (ROP) because of their second-stage exploit (CVE-2010-2883) triggering the PEB heuristic. We verified these attacks used ROP through subsequent manual analysis of the javascript included in the PDFs and reiterate that our current runtime heuristics do not directly detect ROP code, but that in all the examples we observed using ROP, control was always transferred to non-ROP shellcode to perform the primary actions of the attack. We believe that in the future the flexibility of `ShellOS`' ability to load arbitrary process snapshots may be leveraged to correctly execute, detect, and diagnose ROP by iterating the stack pointer (instead of the `IP`) over a buffer and issuing a `ret` instruction to test every position of a buffer for ROP. This may be critical as attackers become more adapt at crafting ROP-only code injection attacks.

Figure 9 depicts the CDF for extracting heap objects from malicious and benign documents. The time distribution for malicious documents is further broken down by "ROP-based" (i.e., CVE-2010-2883) and *other* exploits. The group labeled *other* performed more traditional heap-spray attacks with self-contained shellcode, and is not particularly interesting (at least, from a forensic standpoint). In either case, we were able to extract approximately 98% of the buffers within 26 seconds. For the benign files, extraction took less than 5 seconds for 98% of the documents. The low processing time of the benign case is because the buffers are allocated just once when the PDF is rendered on open, as opposed to hundreds of heap objects created by the embedded javascript that performs the heap-sprays.

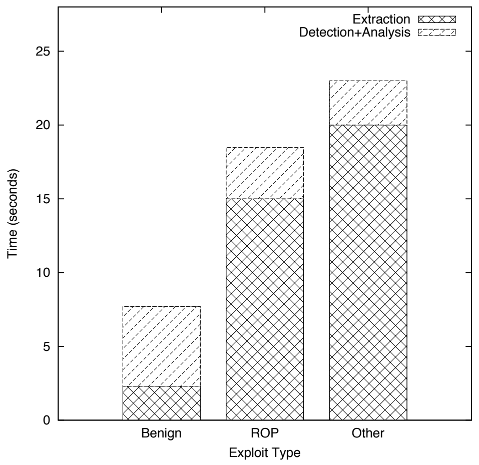

Figure 10: Breakdown of average time of analysis.

The overall time for performing our analyses is given in Figure 10. Notice that the majority of the time can be attributed to buffer extraction. Once signalled, `ShellOS` analyzes the buffers at high speed. The average time to analyze a benign PDF (the common case, hopefully) is 5.46 seconds with our unoptimized code.

We remind the reader that the framework we provide is not tied to any particular method of buffer extraction. To the contrary, ShellOS executes any arbitrary buffer supplied by the analyst and reports if the desired heuristics are triggered. In this case-study, we simply chose to highlight the usefulness of ShellOS with buffers provided by our own PDF pre-processor.

Next, we describe some of the patterns we observed lurking within PDF-based code injection attacks.

5.4 Forensic Analysis

Recall that once injected code is detected, ShellOS continues to allow execution to collect diagnostic traces of Windows API calls before returning a result. In the majority of cases, the diagnostics completed successfully for the PDF dataset. Of the diagnostics performed in the *other* category, we found that 85% of the injected code exhibited an identical API call sequence:

```
begin snippet

LoadLibraryA("urlmon")
URLDownloadToCacheFile(
     URL = "http://(omitted).cz.cc/
                        out.php?a=36&p=5",
     CacheFile = "%tmp%")
CreateProcessA(App = "%tmp%", Cmd = (null))
TerminateThread(Thread = -2, ExitCode = 0)

end snippet
```

The top level domains were always cz.cc and the GET request parameters varied only in numerical value. We also observed that *all* of the remaining PDFs in the *other* category (where diagnostics succeeded) used either the URLDownloadToCacheFile or URLDownloadToFile API call to download a file, then executed it with CreateProcessA, WinExec, or ShellExecuteA. Two of these shellcodes attempted to download several binaries from the same domain, and a few of the requested URLs contained obvious text-based information pertinent to the exploit used, e.g. exp=PDF (Collab), exp=PDF (GetIcon), or ex=Util.Printf – presumably for bookkeeping in an overall diverse attack campaign.

Two of the self-contained payloads were only partially analyzed by the diagnostics, and proved to be quite interesting. The partial call trace for the first of these is given in Figure 11. Here, the injected code allocates space on the heap, then copies code into that heap area. Although the code copy is not apparent in the API call sequence alone, ShellOS may also provide an instruction-level trace (when requested by the analyst) by single-stepping each instruction via the TRAP bit in the flags register. We observed the assembly-level copies using this feature. The code then proceeds to patch several DLL functions, partially observed in this trace by the use of API calls to modify page permissions prior to patching, then resetting them after patching. Again, the assembly-level patching code is only observable in a full instruction trace. Finally, the shellcode performs the conventional URL download and executes that download.

```
begin snippet
GlobalAlloc(Flags = 0x0, Bytes = 8192)
VirtualProtect(Addr = 0x7c86304a, Size = 4096,
               Protect = 0x40)
VirtualProtect(Addr = 0x7c86304a, Size = 4096,
               Protect = 0x20)
LoadLibraryA("user32")
VirtualProtect(Addr = 0x77d702d3, Size = 4096,
               Protect = 0x40)
VirtualProtect(Addr = 0x77d702d3, Size = 4096,
               Protect = 0x20)
LoadLibraryA("ntdll")
VirtualProtect(Addr = 0x7c918c2e, Size = 4096,
               Protect = 0x40)
VirtualProtect(Addr = 0x7c918c2e, Size = 4096,
               Protect = 0x20)
LoadLibraryA("urlmon")
URLDownloadToCacheFile(
     URL = "http://www.(omitted).net/file.exe",
     CacheFile = "%tmp%")
CreateProcessA(App=(null), Cmd="cmd /c %tmp%")
...
end snippet
```

Figure 11: More complex shellcode in a PDF

The second interesting case challenges our prototype diagnostics by applying some anti-analysis techniques. The partial API call sequence observed follows:

```
begin snippet
GetFileSize(hFile = 0x4)
GetTickCount()
GlobalAlloc(Flags = 0x40, Bytes = 4) = buf*
ReadFile(hFile = 0x0, Buf* = buf*, Len = 4)
...continues to loop in this sequence...
end snippet
```

Figure 12: Analysis-resistant Shellcode

As ShellOS does not currently address context-sensitive code, we have no way of providing the file size expected by this code. Furthermore, we do not provide the required timing characteristics for this particular sequence as our API call handlers merely attempt to provide a 'correct' value, with minimal behind-the-scenes processing. As a result, this sequence of API calls is repeated in an infinite loop, preventing further automated analysis. We note, however, that this particular challenge is not unique to ShellOS.

Of the 70 detected ROP-based exploit PDFs, 87% of the second stage payloads adhered to the following API call sequence:

```
——————— begin snippet ———————
LoadLibraryA("urlmon")
LoadLibraryA("shell32")
GetTempPathA(Len = 64, Buffer = "C:\TEMP\")
URLDownloadToFile(
     URL = "http://(omitted).php?
     spl=pdf_sing&s=0907...(omitted)...FC2_1
     &fh=",
     File = "C:\TEMP\a.exe")
ShellExecuteA(File = "C:\TEMP\a.exe")
ExitProcess(ExitCode = -2),
——————— end snippet ———————
```

Figure 13: Typical second stage of a ROP-based PDF code injection attacks observed using ShellOS.

Of the remaining payloads, 6 use an API not yet supported in ShellOS, while the others are simple variants on this conventional URL download pattern.

6 Limitations

Code injection attack detection based on run-time analysis, whether emulated or supported through direct CPU execution, generally operates as a self-sufficient blackbox wherein a suspicious buffer of code or data is supplied, and a result returned. ShellOS attempts to provide a run-time environment as similar as possible to that which the injected code expects. That said, we cannot ignore the fact that shellcode designed to execute under very specific conditions may not operate as expected (e.g., non-self-contained [19, 26], context-keyed [11], and swarm attacks [5]). We note, however, that by requiring more specific processor state, the attack exposure is reduced, which is usually counter to the desired goal — that is, exploiting as many systems as possible. The same rational holds for the use of ROP-based attacks, which require specific data being present in memory.

More specific to our framework is that we currently employ a simplistic approach for loop detection. Whereas software-based emulators are able to quickly detect and (safely) exit an infinite loop by inspecting program state at each instruction, we only have the opportunity to inspect state at each clock tick. At present, the overhead associated with increasing timer frequency to inspect program state more often limits our ability to exit from infinite loops more quickly. In future work, we plan to explore alternative methods for safely pruning such loops, without incurring excessive overhead.

Furthermore, while employing hardware virtualization to run ShellOS provides increased transparency over previous approaches, it may still be possible to detect a virtualized environment through the small set of instructions that must still be emulated. We note, however, that while ShellOS currently uses hardware virtualization extensions to run along side a standard host OS, only implementation of device drivers prevents ShellOS from running directly as the host OS. Running directly as the host OS could have additional performance benefits in detecting code injection for network services. We leave this for future work.

Finally, ShellOS provides a framework for fast detection and analysis of a buffer, but an analyst or automated data pre-processor (such as that presented in §5) must provide these buffers. As our own experience has shown, doing so can be non-trivial, as special attention must be taken to ensure a realistic operating environment is provided to illicit the proper execution of the sample under inspection. This same challenge holds for all VM or emulation-based detection approaches we are aware of (e.g., [6, 8, 10, 31]). Our framework can be extended to benefit from the active body of research in this area.

7 Conclusion

In this paper, we propose a new framework for enabling fast and accurate detection of code injection attacks. Specifically, we take advantage of hardware virtualization to allow for efficient and accurate inspection of buffers by *directly executing* instruction sequences on the CPU. Our approach allows for the modular use of existing run-time heuristics in a manner that does not require tracing every machine-level instruction, or performing unsafe optimizations. In doing so, we provide a foundation that defenses for code injection attacks can build upon. We also provide an empirical evaluation, spanning real-world attacks, that aptly demonstrates the strengths of our framework.

Code Availability

We anticipate that the source code for the ShellOS kernel and our packaged tools will be made available under a BSD license for research and non-commercial uses. Please contact the first author for more information on obtaining the software.

Acknowledgments

We are especially grateful to Michalis Polychronakis for making nemu available to us, and for fruitful discussions regarding this work. Thanks to Teryl Taylor, Scott Coull, Montek Singh and the anonymous reviewers for their insightful comments and suggestions for improving an earlier draft of this paper. We also thank Bil Hayes and Murray Anderegg for their help in setting up the networking infrastructure that supported some of the throughput analyses in this paper. This work is supported by the

National Science Foundation under award CNS-0915364 and by a Google Research Award.

Notes

[1] See, for example, "*Sophisticated, targeted malicious PDF documents exploiting CVE-2009-4324*" at `http://isc.sans.edu/diary.html?storyid=7867`.

[2] See the discussion at `https://bugs.launchpad.net/qemu/+bug/661696`, November, 2010.

[3] We reset registers via `popa` and `fxrstor` instructions, while memory is reset by traversing page table entries and reloading pages with the dirty bit set.

[4] The TAPiON engine is available at `http://pb.specialised.info/all/tapion/`.

[5] We update this network trace with payload byte distributions collected in 2011.

References

[1] P. Baecher and M. Koetter. Libemu - x86 shellcode emulation library. Available at `http://libemu.carnivore.it/`, 2007.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security*, Oct. 2008.

[4] B. Z. Charles Curtsigner, Benjamin Livshits and C. Seifert. Zozzle: Fast and Precise In-Browser Javascript Malware Detection. USENIX Security Symposium, August 2011.

[5] S. P. Chung and A. K. Mok. Swarm attacks against network-level emulation/analysis. In *International symposium on Recent Advances in Intrusion Detection*, pages 175–190, 2008.

[6] M. Cova, C. Kruegel, and V. Giovanni. Detection and analysis of drive-by-download attacks and malicious javascript code. In *International conference on World Wide Web*, pages 281–290, 2010.

[7] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *Annual Computer Security Applications Conference*, pages 327–336, 2010.

[8] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment*, June 2009.

[9] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *USENIX Security Symposium*, pages 241–256, 2006.

[10] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious flash advertisements. In *Computer Security Applications Conference*, pages 363 –372, Dec 2009.

[11] D. A. Glynos. Context-keyed Payload Encoding: Fighting the Next Generation of IDS. In *Athens IT Security Conference (ATH.C0N)*, 2010.

[12] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–35, 1974.

[13] B. Gu, X. Bai, Z. Yang, A. C. Champion, and D. Xuan. Malicious shellcode detection with virtual memory snapshots. In *International Conference on Computer Communications (INFOCOM)*, pages 974–982, 2010.

[14] F. Hernandez-Campos, F. Smith, and K. Jeffay. Tracking the evolution of web traffic: 1995-2003. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS)*, pages 16–25, 2003.

[15] F. Hernandez-Campos, K. Jeffay, and F. Smith. Modeling and generating TCP application workloads. In *14th IEEE International Conference on Broadband Communications, Networks and Systems (BROADNETS)*, pages 280–289, 2007.

[16] I. Kim, K. Kang, Y. Choi, D. Kim, J. Oh, and K. Han. A Practical Approach for Detecting Executable Codes in Network Traffic. In *Asia-Pacific Network Ops. & Mngt Symposium*, 2007.

[17] G. MacManus and M. Sutton. Punk Ode: Hiding Shellcode in Plain Sight. In *Black Hat USA*, 2006.

[18] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *International Symposium on Software Testing and Analysis*, pages 261–272, 2009.

[19] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Conference on Computer and Communications Security*, pages 524–533, 2009.

[20] MSDN. Mindump header structure. MSDN Library. See `http://msdn.microsoft.`

`com/en-us/library/ms680378(VS.85).aspx`.

[21] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *USENIX Workshop on Offensive Technologies*, 2009.

[22] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, R. C. Kuo, and K. P. Fan. Buttercup: on Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *IEEE/IFIP Network Op. & Mngt Symposium*, pages 235–248, May 2004.

[23] U. Payer, P. Teufl, and M. Lamberger. Hybrid Engine for Polymorphic Shellcode Detection. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 19–31, 2005.

[24] J. D. Pincus and B. Baker. Beyond stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security and Privacy*, 4(2):20–27, 2004.

[25] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level Polymorphic Shellcode Detection using Emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 54–73, 2006.

[26] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In *International Symposium on Recent Advances in Intrusion Detection*, 2007.

[27] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[28] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Annual Computer Security Applications Conference*, pages 287–296, 2010.

[29] P. V. Prahbu, Y. Song, and S. J. Stolfo. Smashing the Stack with Hydra: The Many Heads of Advanced Polymorphic Shellcode, 2009. Presented at Defcon 17, Las Vegas.

[30] M. Probst. Fast machine-adaptable dynamic binary translation. In *Proceedings of the Workshop on Binary Translation*, 2001.

[31] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Usenix Workshop on Hot Topics in Botnets*, 2007.

[32] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, pages 1–15, 2008.

[33] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. *Information Security*, 4779:1–18, 2007.

[34] M. A. Rahman. Getting 0wned by malicious PDF - analysis. SANS Institute, InfoSec Reading Room, 2010.

[35] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[36] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections. In *Black Hat USA*, 2008.

[37] D. Stevens. Malicious PDF documents. Information Systems Security Association (ISSA) Journal, July 2010.

[38] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *International Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[39] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, pages 4:1–4:6, New York, NY, USA, 2011.

[40] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *21st Annual Computer Security Applications Conference*, pages 381–392, 2005.

[41] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. *Annual Computer Security Applications Conference*, pages 289–298, Dec 2008.

[42] Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. In *ACM Conference on Computer and Communications Security*, pages 11–20, 2009.

[43] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer. Analyzing Network Traffic to Detect Self-Decrypting Exploit Code. In *ACM Symposium on Information, Computer and Communications Security*, 2007.