2013 IEEE Symposium on Security and Privacy

Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization

Kevin Z. Snow, Fabian Monrose
Department of Computer Science
University of North Carolina at Chapel Hill, USA
Email: {kzsnow,fabian}@cs.unc.edu

Lucas Davi, Alexandra Dmitrienko,
Christopher Liebchen, Ahmad-Reza Sadeghi
CASED/Technische Universität Darmstadt, Germany
Email: {lucas.davi,alexandra.dmitrienko,
christopher.liebchen,ahmad.sadeghi}@trust.cased.de

***Abstract*—Fine-grained address space layout randomization (ASLR) has recently been proposed as a method of efficiently mitigating runtime attacks. In this paper, we introduce the design and implementation of a framework based on a novel attack strategy, dubbed *just-in-time code reuse*, that undermines the benefits of fine-grained ASLR. Specifically, we derail the assumptions embodied in fine-grained ASLR by exploiting the ability to repeatedly abuse a memory disclosure to map an application's memory layout on-the-fly, dynamically discover API functions and gadgets, and JIT-compile a target program using those gadgets—all within a script environment at the time an exploit is launched. We demonstrate the power of our framework by using it in conjunction with a real-world exploit against Internet Explorer, and also provide extensive evaluations that demonstrate the practicality of just-in-time code reuse attacks. Our findings suggest that fine-grained ASLR may not be as promising as first thought.**

I. Introduction

Plus ça change, plus c'est la même chose
— Jean-Baptiste Alphonse Karr (1808-1890).

Today's security landscape paints a disturbing scene. Underground economies that seem to thrive on an unlimited supply of compromised end-user systems (*e.g.*, for monetary gains from a myriad of illicit activities) are as vibrant as ever before. Perhaps more worrying is the trend towards targeting specific entities (*e.g.*, Fortune 500 companies) with valuable intellectual property and trade secrets that are then sold to competitors. Many of these targets are compromised via so-called runtime attacks that exploit vulnerabilities in popular applications (*e.g.*, browsers or document readers).

Despite differences in the style and implementation of these exploits, they all share a common goal: the ability to redirect program logic within the vulnerable application. Sadly, the history of security vulnerabilities enabling these exploits now stretches well into a third decade, and still poses a significant threat to modern systems [59]. Indeed, although numerous defenses have been implemented to limit the scope of these attacks, such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), the cat and mouse games plays on.

In the early days of exploitation, the lack of proper bounds checking was misused to overwrite information on the stack (*e.g.*, a function's return address) and redirect the logical flow of a vulnerable application to arbitrary code (coined *shellcode*), an attack strategy which became known as smashing the stack [3]. To mitigate stack smashing, a so-called *canary* (*i.e.*, a random value) was introduced on the stack preceding the return value, and compilers added a verification routine to function epilogues that terminates programs when the canary is modified [16]. As was to be expected, attackers quickly adapted their exploits by overwriting alternative control-flow constructs, such as structured exception handlers (SEH) [36].

In response, a *no-execute* (NX) bit was introduced into the `x86` architecture's paging scheme that allows any page of memory to be marked as non-executable. DEP leverages the NX bit to mark the stack and heap as non-executable and terminates a running application if control flow is redirected to injected code. Not wanting to be outdone, attackers then added code reuse attacks to their playbook. This new strategy utilizes code already present in memory, instead of relying on code injection. The canonical example is `return-to-libc` [41, 54], in which attacks re-direct execution to existing shared-library functions. More recently, this concept was extended by Shacham [52] to chain together short instruction sequences ending with a `ret` instruction (called *gadgets*) to implement arbitrary program logic. This approach was dubbed *return-oriented programming*. To date, return-oriented programming has been applied to a broad range of architectures (including Intel x86 [52], SPARC [10], Atmel AVR [20], ARM [25, 31], and PowerPC [35]).

This early form of code reuse, however, relies on gadgets being located at known addresses in memory. Thus, address-space layout randomization [19], which randomizes the location of both data and code regions, offered a plausible defensive strategy against these attacks. Code region layout randomization hinders code reuse in exploits; data randomization impedes the redirection of control-flow by making it difficult to guess the location of injected code. Not to be outdone, attackers soon reconciled with an oft neglected class of vulnerabilities: the *memory disclosure*. Indeed, disclosing a single address violates fundamental assumptions in ASLR and effectively reveals the location of

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.45

every piece of code within a single library, thus re-enabling the code reuse attack strategy.

In light of this new code reuse paradigm (whether return-oriented, jump-oriented [9], or some other form of "borrowed code" [32]), skilled adversaries have been actively searching for ever more ingenious ways to leverage memory disclosures (*e.g.*, [33, 51, 56, 60]) as part of their arsenal. At the same time, defenders have been busily working to fortify perimeters by designing "enhanced" randomization strategies [7, 22, 23, 29, 45, 62] for repelling the next generation of wily hackers. In this paper, we question whether this particular line of thinking (regarding fine-grained code randomization) offers a viable alternative in the long run. In particular, we examine the folly of recent exploit mitigation techniques, and show that memory disclosures are far more damaging than previously believed. Just as the introduction of SEH overwrites shattered the illusion of protection provided by stack canaries, code reuse undermines DEP, and memory disclosures defied the basic premise of ASLR, we assail the assumptions embodied by fine-grained ASLR.

Our primary contribution is in showing that fine-grained ASLR for exploit mitigation, even considering an ideal implementation, may not be any more effective than traditional ASLR implementations that are already being bypassed — in short, *the more things change, the more they stay the same*[1]. We provide strong evidence for this by implementing a framework wherein we automatically adapt an arbitrary memory disclosure to one that can be used multiple times to reliably map a vulnerable application's memory layout, then just-in-time compile the attacker's program, re-using (finely randomized) code. The workflow of our framework takes place *entirely* within a single script (*e.g.*, as used by browsers) for remote exploits confronting application-level randomization, or a single binary for local privilege escalation exploits battling kernel-level randomization.

In light of these findings, we argue that the trend toward fine-grained ASLR strategies may be short-sighted. It is our hope that, moving forward, our work spurs discussion and inspires others to explore more comprehensive defensive strategies than what exists today.

II. Background

We first review the basics of important concepts (namely, the code reuse attack strategy and fine-grained memory and code randomization) that are vital to understanding the remainder of this paper.

Code Reuse Attacks: The general principle of any code reuse attack is to redirect the logical program flow to instructions already present in memory, then use those instructions to provide alternative program logic. There exist countless methods of orchestrating such an attack, the simplest of which involves an adversary redirecting the program execution to an existing library function [41, 54]. More generally, Shacham [52] introduced return-oriented programming (ROP) showing that attacks may combine short instruction sequences from *within* functions, called *gadgets*, allowing an adversary to induce *arbitrary* program behavior. Recently, this concept was generalized by removing the reliance on actual return instructions [12]. However, for simplicity, we highlight the basic idea of code reuse using ROP in Figure 1.

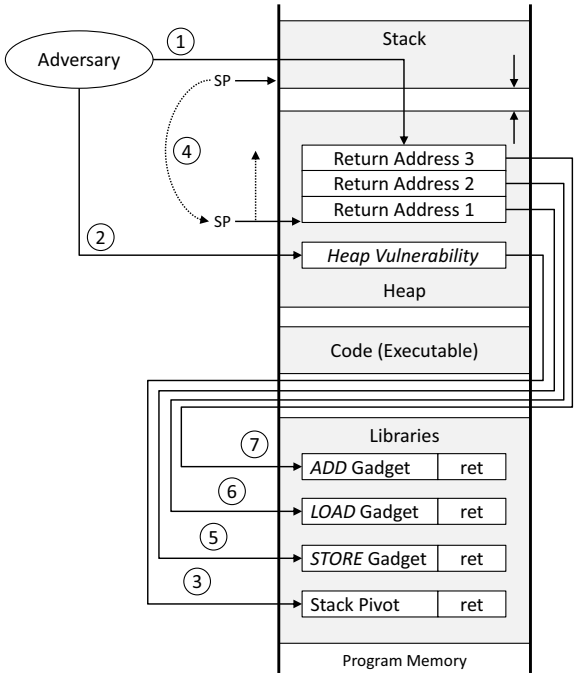

Figure 1. Basic principle of code reuse attacks. For simplicity, we highlight a ROP attack on the heap using a sequence of single-instruction gadgets.

First, the adversary writes a so-called ROP payload into the application's memory space, where the payload mainly consists of a number of pointers (the return addresses) and any other data that is needed for running the attack (Step ①). In particular, the payload is placed into a memory area that can be controlled by the adversary, *i.e.*, the area is writable and the adversary knows its start address. The next step is to exploit a vulnerability of the target program to hijack the intended execution-flow (Step ②). In the example shown in Figure 1, the adversary exploits a heap vulnerability by overwriting the address of a function pointer with an address that points to a so-called *stack pivot* sequence [65]. Once the overwritten function pointer is used by the application, the execution flow is redirected (Step ③).

Loosely speaking, stack pivot sequences change the value of the stack pointer (`%esp`) to a value stored in another

[1]Translation of the original quote, "Plus ça change, plus c'est la même chose," by French novelist Jean-Baptiste Alphonse Karr.

register. Hence, by controlling that register[2], the attacker can arbitrarily change the stack pointer. Typically, the stack pivot directs the stack pointer to the beginning of the payload (Step ④). A concrete example of a stack pivot sequence is the `x86` assembler code sequence `mov %esp,%eax; ret`. The sequence changes the value of the stack pointer to the value stored in register `%eax` and afterwards invokes a return (`ret`) instruction. The `x86 ret` instruction simply loads the address pointed to by `%esp` into the instruction pointer and increments `%esp` by one word. Hence, the execution continues at the first gadget (`STORE`) pointed to by Return Address 1 (Step ⑤). In addition, the stack pointer is increased and now points to Return Address 2.

A gadget represents an atomic operation such as `LOAD`, `ADD`, or `STORE`, followed by a `ret` instruction. It is exactly the terminating `ret` instruction that enables the chained execution of gadgets by loading the address the stack pointer points to (Return Address 2) in the instruction pointer and updating the stack pointer so that it points to the next address in the payload (Return Address 3). Steps ⑤ to ⑦ are repeated until the adversary reaches her goal. To summarize, the combination of different gadgets allows an adversary to induce arbitrary program behavior.

Randomization for Exploit Mitigation: A well-accepted countermeasure against code reuse attacks is the randomization of the application's memory layout. The basic idea of address space layout randomization (ASLR) dates back to Forrest et al. [19], wherein a new stack memory allocator was introduced that adds a random pad for stack objects larger than 16 bytes. Today, ASLR is enabled on nearly all modern operating systems such as Windows, Linux, iOS, or Android. For the most part, current ASLR schemes randomize the base (start) address of segments such as the stack, heap, libraries, and the executable itself. This basic approach is depicted in Figure 2, where the start address of an executable is relocated between consecutive runs of the application. As a result, an adversary must guess the location of the functions and instruction sequences needed for successful deployment of her code reuse attack.

Unfortunately, today's ASLR realizations suffer from two main problems: first, the entropy on 32-bit systems is too low, and thus ASLR can be bypassed by means of brute-force attacks [37, 53]. Second, all ASLR solutions are vulnerable to *memory disclosure* attacks [51, 56] where the adversary gains knowledge of a single runtime address and uses that information to re-enable code reuse in her playbook once again. Many of today's most sophisticated exploits use *JavaScript* or *ActionScript* (hereafter referred to as a *script*) and a memory-disclosure vulnerability to reveal the location of a single code module (*e.g.*, a dynamically-loaded library) loaded in memory. Since current ASLR implementations only randomize on a per-module level, disclosing a single address within a module effectively reveals the location of every piece of code within that module. Therefore, any gadgets from a disclosed module may be determined manually by the attacker offline prior to deploying the exploit. Once the prerequisite information has been gathered, the exploit script simply builds a payload from a pre-determined template by adjusting offsets based on the module's disclosed location at runtime.

[2]To control the register, the adversary can either use a buffer overflow exploit that overwrites memory areas that are used to load the target register, or invoke a sequence that initializes the target register and then directly calls the stack pivot.

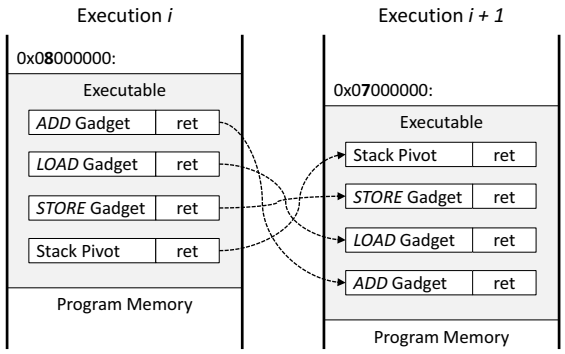

Figure 2. Fine-Grained Memory and Code Randomization

To confound these attacks, a number of fine-grained ASLR and code randomization schemes have recently appeared in the academic literature [7, 23, 29, 45, 62]. We elaborate on these techniques later (in §VI), but for now it is sufficient to note that the underlying idea in these works is to randomize the data and code structure, for instance, by shuffling functions or basic blocks (ideally for each program run [62]). As shown in Figure 2, the result of this approach is that the location of all gadgets is randomized. The assumption underlying all these works is that the disclosure of a single address no longer allows an adversary to deploy a code reuse attack.

III. Assumptions and Adversarial Model

We now turn to our assumptions and adversarial model. In general, an adversary's actions may be enumerated in two stages: (1) exercise a vulnerable entry point, and (2) execute arbitrary malicious computations. Similar to previous work on runtime attacks (*e.g.*, the original paper on return-oriented programming [52]), our assumptions cover defense mechanisms for the second stage of runtime attacks, *i.e.*, the execution of malicious computations. Modern stack and heap mitigations (such as heap allocation order randomization) do eliminate categories of attack supporting stage one, but these mitigations are not comprehensive (*i.e.*, exploitable vulnerabilities still exist). Thus, we assume the adversary is able to exercise one of these pre-existing vulnerable entry points. Hence, a full discussion of the first stage of attack is out of scope for this paper. We refer the interested

reader to [26] for an in-depth discussion on stack and heap vulnerability mitigations.

In what follows, we assume that the target platform uses the following mechanisms (see §II) to mitigate the execution of malicious computations:

- **Non-Executable Memory:** We assume that the security model of non-executable memory (also called NX or DEP) is applied to the stack and the heap. Hence, the adversary is not able to inject code into the program's data area. Further, we assume that the same mechanism is applied to all executables and native system libraries, thereby preventing one from overwriting existing code.
- **JIT Mitigations:** We assume a full-suite of JIT-spraying mitigations, such as randomized JIT pages, constant variable modifications, and random NOP insertion. As our approach is unrelated to JIT-spraying attacks, these mitigations provide no additional protection against our code reuse attack.
- **Export Address Table Access Filtering:** We assume code outside of a module's code segment cannot access a shared library's export table (*i.e.* as commonly used by shellcode to lookup API function addresses). As our approach applies code-reuse from existing modules for 100% of malicious computations, this mitigation provides no additional protection.
- **Base Address Randomization:** We assume that the target platform deploys base address randomization by means of ASLR and that all useful, predictable, mappings have been eliminated.
- **Fine-Grained ASLR:** We assume that the target platform enforces fine-grained memory and code randomization on executables and libraries. In particular, we assume a strong fine-grained randomization scheme, which (i) permutes the order of functions [7, 29] and basic blocks [62], (ii) swaps registers and replaces instructions [45], (iii) randomizes the location of each instruction [23], and (iv) performs randomization upon each run of an application [62].

Notice that our assumptions on deployed protection mechanisms go beyond what current platforms typically provide. For instance, ASLR is not usually applied to every executable or library, thereby allowing an adversary to leverage the non-randomized code parts for a conventional code reuse attack. Further, current systems do not enforce fine-grained randomization. That said, there is a visible trend toward enabling ASLR for all applications, even for the operating system kernel (as recently deployed in Windows 8). Furthermore, current thinking is that fine-grained randomization has been argued to be efficient enough to be considered as a mechanism by which operating system vendors can tackle the deficiencies of base address randomization.

Nevertheless, even given all these fortified defenses, we show that our framework for code reuse attacks can readily undermine the security provided by these techniques. In fact, an adversary utilizing our framework—whether bypassing ASLR or fine-grained mitigations—will enjoy a simpler and more streamlined exploit development process than ever before. As will become apparent in the next section, our framework frees the adversary from the burden of manually piecing together complicated code reuse payloads and, because we build the entire payload on-the-fly, it can be made compatible with all OS revisions. We only assume that the adversary can (1) conform a memory disclosure vulnerability to our interface that reveals values at an absolute address, and (2) discover a single code pointer, *e.g.*, as typically found via function pointers described by a heap or stack-allocated object. We find these assumptions to be quite practical, as existing exploits that bypass standard ASLR have nearly identical requirements (see *e.g.*, [50]).

IV. Overview of Just-In-Time Code Reuse

Unfortunately, nearly all fine-grained exploit mitigations to-date fail to precisely define an adversarial model or are based on a model that fails to consider *multiple* memory-disclosures. This is problematic for a number of reasons, the least of which is that discrediting multiple memory disclosures is impractical. Our key observation is that by exploiting a memory disclosure multiple times we violate implicit assumptions of the fine-grained exploit mitigation model and enable the adversary to iterate over mapped memory to search for all necessary gadgets on-the-fly, regardless of the granularity of code and memory randomization. It is our conjecture that if fine-grained exploit mitigations became common-place, then attackers would simply modularize, automate, and incorporate a similar approach into existing exploitation toolkits (*e.g., metasploit* [39]).

To evaluate the hypothesis that multiple memory disclosures are an effective means to bypass fine-grained exploit mitigation techniques, we have designed and built a prototype exploit framework that aptly demonstrates one instantiation (called `JIT-ROP`) of our idea. The overall workflow of an exploit using our framework is given in Figure 3. An adversary constructing a new exploit need only conform their memory disclosure to our interface and provide an initial code pointer in Step ❶, then let our framework take over in Steps ❷ to ❺ to automatically (and at exploit runtime) harvest additional code pages, find API functions and gadgets, and just-in-time compile the attacker's program to a serialized payload useable by the exploit script in Step ❻.

The implementation was highly involved, and successful completion overcame several challenges. With that in mind, we also remind the reader that the current implementation of our framework represents but *one* instantiation of a variety of advanced techniques that could be applied at each step, *e.g.*, one could add support for jump-oriented programming [9], use more advanced register allocation schemes

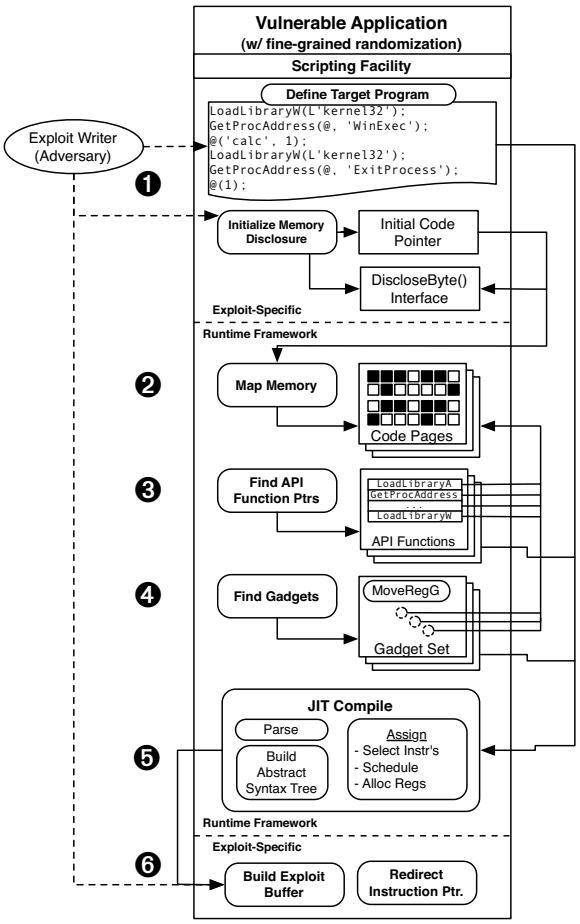

Figure 3. Overall workflow of a code injection attack utilizing just-in-time code reuse against a script-enabled application protected by fine-grained memory (or code) randomization.

from compiler theory, etc. Nevertheless, we show in §V that our implementation of JIT-ROP is more than sufficient for real-world deployment, both in terms of stability and performance. In the remainder of this section, we elaborate on the necessary components of our system, and conclude with a concrete example of an exploit using our framework.

A. Mapping Code Page Memory

Prior to even considering a code reuse attack, the adversary must be made aware of the code already present in memory. The first challenge lies in developing a reliable method for automatically searching through memory for code without causing a crash (*e.g.*, as a result of reading an unmapped memory address). On 32-bit Microsoft Windows, applications may typically address up to 3 GB, while on 64-bit Microsoft Windows this can be several terabytes. Applications, however, typically use less than a few hundred megabytes of that total space. Thus, simply guessing addresses to disclose is likely to crash the vulnerable application. Furthermore, our assumptions in §III forbid us from relying on any prior knowledge of module load addresses, which would be unreliable to obtain in face of fine-grained ASLR and non-continuous memory regions.

To overcome this hurdle, we note that knowledge of a single valid code pointer (*e.g.*, gleaned from the heap or stack) reveals that an entire 4 kilobyte-aligned page of memory is guaranteed to be mapped. In Step ❶, we require the exploit writer to conform the single memory disclosure to an interface named `DiscloseByte` that, given an absolute virtual address, discloses one byte of data at that address (see §IV-E). One approach, therefore, is to use the `DiscloseByte` method to implement a `DisclosePage` method that, given an address, discloses an entire page of memory data. The challenge then is to enumerate any information found in this initial page of code that reliably identifies additional pages of code.

One reliable source of information on additional code pages is contained within the control-flow instructions of the application itself. Since code typically spans thousands of pages, there must be control-flow links between them. In Step ❷, we apply static code analysis techniques (in our case, at runtime) to identify both direct and indirect `call` and `jmp` control-flow instructions within the initial code page. We gather new code pages from instructions disassembled in the initial code page. Direct control-flow instructions yield an immediate hint at another code location, sometimes in another page of memory. Indirect control-flow instructions, on the other hand, often point to other modules (*e.g.*, as in a call to another DLL function), and so we can process these by disclosing the address value in the Import Address Table (IAT) pointed to by the indirect instruction (*i.e.*, we implement a `DisclosePointer` method on top of `DiscloseByte`). In practice, we may be able to derive additional code pointers from other entries found around the address disclosed in the IAT, or read past page boundaries if code regions are contiguous. However, we choose not to use this information as our assumptions on the application of ideal fine-grained randomization forbid us from doing so.

By applying this discovery technique iteratively on each code page found, we can map a significant portion of the code layout of the application instance's virtual address space. Algorithm 1 is a recursive search over discovered code pages that results in the set of unique code page virtual addresses along with associated data. The `Disassemble` routine performs a simple linear sweep disassembly over the entirety of the code page data. As compilers sometimes embed data (*e.g.* a jump table) and padding (*e.g.* between functions) within code, disassembly errors may arise. More deliberate binary obfuscations may also be used by a vulnerable program to make this step more difficult, although obfuscations of this magnitude are not typically applied by reputable software vendors. To minimize these errors, however, we filter out both invalid (or privileged) instructions and valid instructions in the immediate vicinity of

Algorithm 1 `HarvestCodePages`: given an initial code page, recursively disassemble pages and discover direct and indirect pointers to other mapped code pages.

Input: P {initial code page pointer}, C {visited set}
Output: C {set of valid code pages}
if $\exists(P \in C)$ {already visited} **then**
  return
end if
$C(P) \leftarrow true$ {Mark page as visited}
$\vec{P}$ = `DisclosePage`(P) {Uses `DiscloseByte()` internally to fetch page data}
for all $ins \in$ `Disassemble`$(\vec{P})$ **do**
  if `isDirectControlFlow`(ins) **then**
    {*e.g.* `JMP +0xBEEF`}
    $ptr \leftarrow ins.offset + ins.effective\_address$
    `HarvestCodePages`(ptr)
  end if
  if `isIndirectControlFlow`(ins) **then**
    {*e.g.* `CALL [-0xFEED]`}
    $iat\_ptr \leftarrow ins.offset + ins.effective\_address$
    $ptr \leftarrow$ `DisclosePointer`(iat_ptr) {Internally uses `DiscloseByte()` to fetch pointer data}
    `HarvestCodePages`(ptr)
  end if
end for

any invalid instruction. While more advanced static analysis could certainly be employed (*e.g.* recursive descent), our approach has proven effective in all our tests.

In short, as new code pages are discovered, static code analysis can be applied to process and store additional unique pages. Iteration continues only until all the requisite information to build a payload has been acquired. That said, automatically building a practical payload requires that we obtain some additional information, which we elaborate on in the next sections.

B. API Function Discovery

The second challenge lies in the fact that an exploit will inevitably need to interact with operating system APIs to enact any significant effect. The importance of this should not be understated: while Turing-complete execution is not needed for a practical payload that reuses code, specialized OS interoperability is required. One method of achieving this is through direct interaction with the kernel via interrupt (`int 0x80`) or fast (`syscall`) system call instruction gadgets. Payloads using hardcoded system calls, however, have been historically avoided because of frequent changes between even minor revisions of an OS. The favored method of interacting with the OS is through API calls (*e.g.*, as in `kernel32.dll`), the same way benign programs interact, because of the relative stability across OS revisions.

Thus, in Step ❸, we must discover the virtual addresses of API functions used in the attacker-supplied program. Past exploits have managed this by parsing the Process Environment Block (`PEB`), as in traditional shellcode, or by manually harvesting an application-specific function pointer from the heap, stack, or IAT in ROP-based exploits. While one may consider emulating the traditional `PEB` parsing strategy using gadgets (rather than shellcode), this approach is problematic because gadgets reading the `PEB` are rare, and thus cannot be reliably applied to code reuse attacks. Moreover, requisite function pointers typically used in ROP-style exploits may not be present on the stack or heap, and the IAT may be randomized by fine-grained exploit mitigations.

Our code page harvesting in Step ❷ gives us unfettered access to a large amount of application code, which presents a unique opportunity for automatically discovering a diverse set of API function pointers. Luckily, API functions desirable in exploit payloads are frequently used in application code and libraries. On Windows, for example, applications and libraries that re-use code from shared libraries usually obtain pointers to the desired library functions through the combination of `LoadLibrary` (to get the library base address) and `GetProcAddress` (to get the function address) functions. In this way, a running process is able to easily obtain the runtime address of any API function. By finding these API functions, we would only need to accurately initialize the arguments (*i.e.*, the strings of the desired library and function) to `LoadLibrary` and `GetProcAddress` to retrieve the address of any other API function.

Our approach to find these API functions is to create signatures based on opcode sequences for call sites of the API functions, then match those signatures to call sites in the pages we traverse at run-time. Our prototype implementation uses static signatures, which we generate a-priori, then match at runtime during code page traversal. This implementation would fail to work if individual instruction transformations were applied to each call site. However, just as antivirus products use *fuzzy* or *semantic* signatures, we too may utilize such a technique if code has been metamorphosed in this way by fine-grained exploit mitigations. Once a call site has been identified, we use `DisclosePointer` to reveal the API function pointer in the IAT and maintain the unique set of functions identified.

C. Gadget Discovery

Thus far we have automatically mapped a significant portion of the vulnerable application's code layout and collected API function pointers required by the exploit writer's designated program. The next challenge lies in accumulating a set of concrete gadgets to use as building blocks for the just-in-time code reuse payload. Since fine-grained exploit mitigations may metamorphose instructions on each execution, we do not have the luxury of searching for gadgets offline and hardcoding the requisite set of gadgets for our payload. Hence, in contrast to previous works on offline gadget compilers [24, 48], we must perform the gadget discovery at the same time the exploit runs. Moreover, we must do this as efficiently as possible in practice because Steps ❶ through ❻ must all run in real-time on the victim's machine. As time-

Algorithm 2 `VerifyGadget`: Automatically match a sequence of instructions to a gadget's semantic definition.

Input: S {sequence of consecutive instructions}, D {gadget semantic definitions}
Output: G {gadget type, or *null*}
$head \leftarrow S(0)$ {first instruction in sequence}
if $G \leftarrow$ `LookupSemantics`$(head) \notin D$ {implemented as a single table-lookup} **then**
  return $null$
end if
for $i \in 1...|S|$ {ensure semantics are not violated by subsequent instructions} **do**
  $ins \leftarrow S(i)$
  if `HasSideEffects`(ins) || `RegsKilled`$(ins) \in$ `RegsOut`$(head)$ **then**
    return $null$
  end if
end for
return G {valid, useful gadget}

Gadget Type Name	Semantic Definition	Example
MovRegG	$OutReg \leftarrow InReg$	mov edi,eax
LoadRegG	$OutReg \leftarrow$ const	pop ebx
ArithmeticG	$OutReg \leftarrow InReg1 \diamond InReg2$	add ecx,ebx
LoadMemG	$OutReg \leftarrow M[InReg]$	mov eax,[edx+0xf]
ArithmeticLoadG	$OutReg \diamond\!\leftarrow M[InReg]$	add esi,[ebp+0x1]
StoreMemG	$M[InReg1] \leftarrow InReg2$	mov [edi],eax
ArithmeticStoreG	$M[InReg1] \diamond\!\leftarrow InReg2$	sub [ebx],esi
StackPivotG	$ESP \leftarrow InReg$	xchg eax,esp
JumpG	$EIP \leftarrow InReg$	jmp edi
NoOpG	$NoEffect$	(ret)

Table I
SEMANTIC DEFINITIONS FOR GADGET TYPES USED IN OUR FRAMEWORK. THE $\diamond$ SYMBOL DENOTES ANY ARITHMETIC OPERATION.

of-exploit increases, so does the possibility of the victim (or built-in application watchdog) terminating the application prior to exploit completion.

Unfettered access to a large number of the code pages enables us to search for gadgets not unlike an offline approach would[3], albeit with computational performance as a primary concern. Thus, in Step ❹ we efficiently collect sequences of instructions by adapting the `Galileo` algorithm proposed by Shacham [52] to iterate over the harvested code pages from Step ❷ and populate an instruction prefix tree structure. As instruction sequences are added to the prefix tree, they are tested to indicate whether they represent a useful gadget. Our criteria for useful gadgets is similar to Schwartz et al. [48], wherein we bin gadgets into types with a unique semantic definition. Table I provides a listing of the gadget types we use, along with their semantic definitions. Higher-order computations are constructed from a composite of these gadget type primitives during compilation. For example, calling a Windows API function using position-independent code-reuse may involve a `MovRegG` to get the value of the stack pointer, an `ArithmeticG` to compute the offset to a pointer parameter's data, a `StoreMemG` or `ArithmeticStoreG` to place the parameter in the correct position on the stack, and a `JumpG` to invoke the API call.

Unlike semantic verification used in prior work, we avoid complex program verification techniques like weakest pre-condition, wherein a theorem prover is invoked to verify an instruction sequence's post-condition meets the given semantic definition [48]. We also discard the idea of so-called concrete execution, wherein rather than proving a post-condition is met, the instructions in the sequence are emulated and the simulated result is compared with the semantic definition. These methods are simply too heavy-weight to be considered in a runtime framework like ours that is interpreted in a script-based environment.

Instead, we opt for a heuristic (Algorithm 2) based on the observation that a single instruction type (or sub-type) fulfills a gadget's semantic definitions. For example, any instruction of the form `mov r32, [r32+offset]` meets the semantic definition of a `LoadMemG` gadget. Thus, for any instruction sequence we perform a single table lookup to check if the first instruction type in the sequence meets one of the gadget semantic definitions. If it does, we add the gadget to our unique set only if the initial instruction's $OutReg$ is not nullified by subsequent instructions, and those instructions do not have adverse side-effects such as accessing memory from an undefined register or modifying the stack pointer. In practice, we found that this heuristic finds mostly the same gadget instruction sequences as the program verification approach.[4]

D. Just-In-Time Compilation

The final challenge lies in using the dynamically discovered API function pointers and collection of concrete gadgets to satisfy the exploit writer's target program (Step ❶ of Figure 3), then generate a payload to execute (Step ❻). Since each instantiation of a vulnerable application may yield a completely different set of concrete gadgets when fine-grained exploit mitigations are used, a dynamic compilation is required to ensure we can use a plethora of gadget types to build the final payload. Fundamentally, our just-in-time gadget compiler is like a traditional compiler, except that compilation is embedded directly within an exploit script, and we only have a subset of concrete instructions (and register combinations) available for code generation. We use syntax directed parsing to process the exploit writer's program, which is written to conform to a simple custom grammar (see Step ❶ for example).

Our grammar supports arbitrary sequences of API function calls with an arbitrary number of parameters that may be static, dynamic, or pointers to data embedded in the payload. Pointers to data (such as 'kernel32') are implemented with position independent gadgets. Using these primitives we are

[3]See offline gadget tools such as `mona` (http://redmine.corelan.be/projects/mona) or `ropc` (http://github.com/pakt/ropc).

[4]Based on sample gadget output provided by Schwartz et al. [48] and available at http://plaid.cylab.cmu.edu:8080/~ed/gadgets/

able to support a variety of payload types, including equivalents to Metasploit's code injection-style execute, download and execute, and message box payloads. An abstract syntax tree (AST) is extended for each program statement, wherein each node corresponds to a set of gadget types (each with their own child edges), any of which can implement the current grammar rule in terms of a tree of gadget types represented by AST nodes. Therefore, each node is actually a set of AST subtrees, any of which performs the same operation, but with different gadgets. We use this structure to efficiently perform a lazy search over all possible gadget combinations that implement a program statement, as well as a search over all schedules and register combinations. To do so, we adapted the algorithms from Schwartz et al. [48] to suit our AST data structure.[5]

Lastly, the final payload is serialized to a structure accessible from the script, and control is returned to the exploit writer's code (Step ❻). As some individual gadgets that are not part of the final payload are likely required to build the exploit buffer (*e.g.*, `StackPivotG`, `NoOpG`), our framework also makes those available to the exploit writer from the gadget collection. Next, we describe how we architected the overall implementation, which turned out to be a significant engineering challenge of its own.

E. Implementation

By now, the astute reader must have surely realized that accomplishing this feat requires a non-trivial architectural design, whereby we disassemble code, recursively traverse code pages, match API function signatures, build an instruction prefix tree and semantically verify gadgets, and just-in-time compile gadgets to a serialized payload useable by the exploit—all performed at runtime in a script environment. While true, we can use modern compiler infrastructures to help with most of the heavy lifting. Figure 4 depicts the overall architecture of our implementation of `JIT-ROP`. The framework is $\sim$ 3000 lines of new `C++` code, and additionally uses the `libdasm` library as a 3rd-party disassembler.

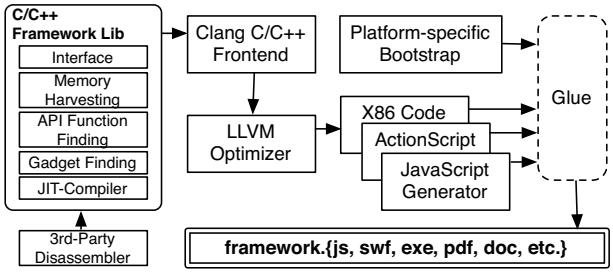

Figure 4. Overall architecture. Since we have no platform-specific dependencies, we can support multiple platforms.

[5]While Schwartz et al. [48] released source code, the `ML` language it is written in is incompatible with our framework and AST data structure. For our purposes, we reimplemented the approach suggested in their paper.

We use the `Clang` front-end to parse our code, and then process it with the `LLVM` compiler infrastructure to generate library code in a variety of supported output formats. Obviously, we can generate an x86 (or ARM) version, as well as ActionScript and JavaScript. These scripting languages cover support for the vast majority of today's exploits, but we could easily generate the framework library for any output format supported by `LLVM`. The native binary versions could also be used to locally bypass kernel-level fine-grained exploit mitigations. Each outputted format needs a small amount of bootstrap code. For example, x86 code needs an executable to use the framework, and our JavaScript output required a small amount of interfacing code ($\sim$ 230 lines). For the purposes of our evaluation in the next section, the end result of our compilation process is a single JavaScript file that could be included in any `HTML` or document-format supporting JavaScript, such as Adobe's PDF Reader.

Proof of Concept Exploit: To demonstrate the power of the framework, we used it to exploit Internet Explorer (IE) 8 running on Windows 7 using CVE-2012-1876. The vulnerability is used to automatically load the Windows `calc` application upon browsing a HTML page. We choose this vulnerability because a technical analysis [60] provides details on construction of an exploit that bypasses ASLR. Since our framework has similar requirements (albeit with more serious implications), construction is straight-forward. Nevertheless, as we describe in §VII, there are a number of other IE vulnerabilities that we could have exploited to develop a proof of concept exploit. The various steps in Figure 3 were accomplished as follows.

First, we setup the memory disclosure interface and reveal an initial code pointer (Step ❶). To do so, we apply heap Feng Shui [55] to arrange objects on the heap in the following order: (1) buffer to overflow, (2) string object, (3) a button object (specifically, `CButtonLayout`). Next, we perform an overflow to write a value of 2^{32} into the string object's length. This allows us to read bytes at any relative memory address (*e.g.*, using the `charCodeAt()` function). We use the relative read to harvest a self-reference from the button object, and implement a `DiscloseByte` interface that translates a relative address to an absolute address. The JavaScript code sample in Figure 5 begins with the implementation of `DiscloseByte` on line 6.

Following that, we define our target program (which launches `calc` with 100% code reuse) in a simple high-level language on lines 15-21. Note that '@' is a shorthand for 'the last value returned' in our `JIT-ROP` language, which may be referenced as a variable or called as a function. Next, the first function pointer harvested from the button object is given to `HarvestCodePages` (line 24), which automatically maps code pages (Step ❷), finds the `LoadLibrary` and `GetProcAddress` functions (Step ❸), and discovers gadgets (Step ❹).

Exploit Code Sample

```
// ... snip ...                                                    1
// The string object is overwritten, and initial code             2
// pointer harvested prior to this snippet of code                3
                                                                  4
// Step 1, implement DiscloseByte interface                       5
framework.prototype.DiscloseByte = function(address) {           6
  var value = this.string_.charCodeAt(                            7
      (address - this.absoluteAddress_ - 8)/2);                   8
  if (address & 1) return value >> 8;    // get upper            9
  return value & 0xFF; // value is 2 bytes, get lower            10
};                                                                11
                                                                  12
// Define target program ('@' is shorthand                        13
// for 'last value returned')                                     14
var program =                                                     15
  "LoadLibraryW(L'kernel32');" +                                  16
  "GetProcAddress(@, 'WinExec');" +                               17
  "@('calc', 1);" +                                               18
  "LoadLibraryW(L'kernel32');" +                                  19
  "GetProcAddress(@, 'ExitProcess');" +                           20
  "@(1);";                                                        21
                                                                  22
// Steps 2-4, harvest pages, gadgets, functions                   23
framework.HarvestCodePages(this.initialCodePtr_);                 24
                                                                  25
// Step 5, 6 - jit-compile and build exploit buffer               26
var exploitBuffer =                                               27
  repeat(0x3E, unescape("%u9191%u9191")) + // Id                  28
  repeat(0x19, framework.NoOpG()) +        // Sled                29
  unescape(framework.Compile(program)) +   // Payload             30
  repeat(0x12, unescape("%u4545%u4545")) + // Pad                 31
  repeat(0x32, framework.StackPivotG());   // Redirect            32
                                                                  33
// overwrite with the exploit buffer                              34
// ... snip ...                                                   35
```

End Code

Figure 5. A JavaScript code sample from our proof of concept exploit illustrating each of the steps from our workflow.

Finally, we JIT-compile the target program (Step ❺, line 30) inline with exploit buffer construction. We redirect program flow (Step ❻) using the same heap layout as in Step ❶. As a result, one of button object's function pointers is overwritten with the stack pivot constructed in line 32 of the exploit buffer. The `StackPivotG` switches the stack to begin execution of the `NoOpG` gadget sled, which in turn begins execution of our program (lines 15-21) that was JIT-compiled to a series of gadgets in line 30.

V. Evaluation

We now evaluate the practicality of our framework by using it in conjunction with our real-world exploit against Internet Explorer (described in §IV) in Windows 7 and a number of other applications. We also provide empirical evaluations of components ❷ through ❻ in our just-in-time code reuse framework.

A. On Code Page Harvesting

The success of our code reuse framework hinges on the ability to dynamically harvest memory pages consisting of executable code, thereby rendering fine-grained randomization ineffective. As alluded to earlier, the adversary provides a starting point for the code page harvesting algorithm by supplying an initial pointer. In the proof of concept, we accomplished this via the `CButtonLayout` object's function pointers on the heap. Starting from this initial page, we harvested 301 code pages from the Internet Explorer process (including those pages harvested from library modules). However, since the final set of harvested pages differ based on the initial page, we opted for an offline empirical evaluation to more thoroughly analyze our performance.

The offline evaluation allows us to test initial code pages that would not normally be found by our proof of concept, since other exploits may indeed begin harvesting with those pages. To perform the evaluation, we use memory *snapshots* created using a custom library. Our library enables us to attach to an application process and store memory contents using the functionality provided by the Windows debug library (`DbgHelp`). The snapshots contain all process memory, metadata indicating if a page is marked as executable code, and auxiliary information on which pages belong to the application or a shared library. The native `x86` version of the framework (see §IV-E) is used to load the snapshots and test the effectiveness of `HarvestCodePages` (Algorithm 1) by independently initializing it from each individual code page within the snapshot. Since using snapshots gives us the ability to evaluate applications without requiring an exploit-in-hand to setup a memory disclosure, we are able to analyze the framework's performance from many angles.

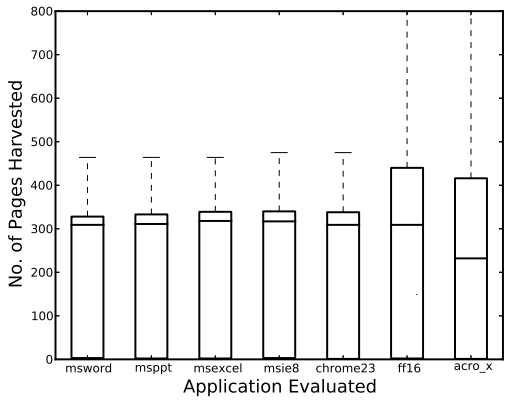

Figure 6. Box and Whisker plot showing the number of unique code pages harvested from different initial pages.

The boxplot in Figure 6 depicts our results on different popular applications. Each box represents thousands of runs of `HarvestCodePage`, where each run uses a different initial code page for harvesting. The results for Internet Explorer (msie8), for example, indicate that for over half of the initial starting points we harvest over 300 pages (*i.e.*, nearly 2MB of code). The top 25th percentile for FireFox (ff16) and Adobe Acrobat Pro X (acro_x) harvest over 1000 pages. The logical explanation for this variability is two-fold. First, code that is well connected (*i.e.*, uses many API calls) naturally allows us to achieve better coverage. Second, compilers sometimes insert data and padding into code segments (*e.g.*,

specialized arrays for compiler optimizations). Therefore, if pages containing such data are used as initialization points, the final set of pages will be lower since there would likely be fewer calls that index into the IAT in those regions.

Module	No. Pages (% of library)	No. Gadgets
gdi32.dll	36 (50%)	31
imm32.dll	16 (69%)	22
kernel32.dll	52 (26%)	61
kernelbase.dll	12 (17%)	22
lpk.dll	5 (83%)	0
mlang.dll	5 (16%)	8
msvcrt.dll	5 (3%)	6
ntdll.dll	109 (50%)	205
user32.dll	47 (45%)	57
uxtheme.dll	20 (35%)	23

Table II
LOCATION OF CODE PAGES HARVESTED IN A SINGLE-RUN OF HarvestCodePages ON INTERNET EXPLORER.

To gain additional insight on exactly what libraries the harvested code pages were from, we used the auxiliary information in our memory snapshot to map the coverage of a single average-case run (307 pages harvested total) of HarvestCodePages on Internet Explorer. Table II enumerates results of the analysis. These results vary depending on the initial code pointer used, but this single instance serves to highlight the point that we harvest pages from a variety of well-connected libraries.

B. On Gadget Coverage

Obviously, the number of recovered code pages is only meaningful if the pages contain usable gadgets. To evaluate how well our gadget discovery process works, we examine the number of gadgets of each type found in a particular set of harvested code pages. While we are able to find all the gadgets (spanning 5 to 6 gadget types) required in our proof of concept exploit, we also demonstrate that regardless of the initial code page used by a particular exploit, we still find enough gadgets to build a payload. In fact, we found that we could generate a payload from 78% of the initial code pages, and 67% of the initial starting points additionally yielded a StackPivotG, which is required for many exploits.

Figure 7 depicts the number of each type of gadget discovered across multiple runs in an offline evaluation. For brevity, we only show the results from Internet Explorer. Each of these runs is initialized with one of the 7,398 different code pages in the snapshot. Gadgets were only considered that (1) had at most 5 instructions in the sequence (excluding RET), and (2) pop at most 40 bytes off the stack. The pop-limit is a matter of practicality—for each additional byte popped, the final payload needs an equivalent amount of padding that increases payload size. We argue that 40 bytes (or 10 gadget slots) is a reasonable threshold, but this value can be adjusted to accommodate other exploits.

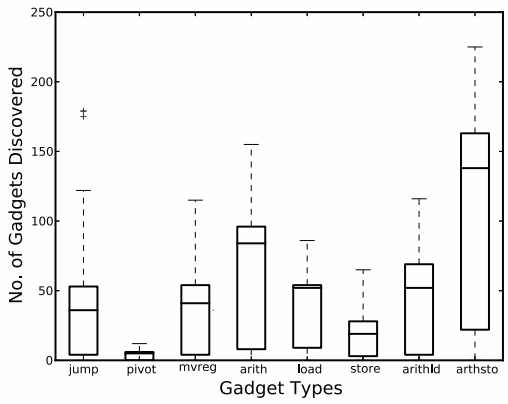

Figure 7. The number of gadgets (of each type) discovered in an Internet Explorer 8 process as we explore from 7,398 distinct starting pages. NoOpG and LoadRegG gadget types are not displayed due to their abundance.

To reinforce the point that gadget discovery is not hindered by fine-grained mitigation techniques, we conducted an experiment using the in-place binary code randomizer (called ORP [45])[6]. ORP was designed as a practical (*i.e.*, it does not require access to source code or debug symbols) defense against code reuse attacks. ORP randomizes instructions within a basic block by reordering or replacing instructions with various narrow-scope code transformations. Thus, a gadget set created using the adversary's instance of an application will be different than the victim's instance—thereby thwarting traditional code reuse attacks. Pappas et al. [45] show they effectively eliminate about 10%, and probabilistically break about 80%, of useful instruction sequences via their code transformation technique.

We attempted to use ORP to randomize a number of applications, but unfortunately were unable to run those applications afterwards due to runtime exceptions[7]. However, we were nevertheless able to use our memory snapshotting facility to instrument a test. To do so, we created a test program that simply loads any given set of libraries (via LoadLibrary). We used a subset of DLLs commonly loaded by Internet Explorer that ORP was able to successfully randomize. There were 52 such DLLs in total. One snapshot was taken on a run loading randomized DLLs, while the other used the original unmodified DLLs. We then ran our offline evaluation on both scenarios (omitting any unrandomized system DLLs). Ironically, our framework discovers slightly more gadgets in the randomized libraries than the original unmodified DLLs, as code that ORP adjusts may inadvertently add new gadgets as old gadgets are eliminated. Our success did not come as a surprise since we discover

[6]We used ORP v0.2 at http://nsl.cs.columbia.edu/projects/orp/orp-0.2.zip.

[7]While the authors of ORP are aware of the issues we encountered, and are working with us on a solution, they were unfortunately unable to resolve the problem before the camera ready version.

gadgets on-the-fly and can therefore find even transformed or newly introduced gadgets — unlike the offline gadget discovery tools against which ORP was originally evaluated.

C. On API Function Discovery

Without API calls to interact with the OS, a JIT-ROP payload would be nothing more than a toy example. Today, it is commonly assumed that the most direct way to undermine non-executable memory is by calling VirtualProtect in a ROP exploit, then transferring control to a second-stage payload composed of traditional shellcode. However, to our surprise, we found that within the Internet Explorer 8 process memory (including all libraries), there were only 15 distinct call sites to VirtualProtect. Therefore, searching for a call-site of this function in face of fine-grained ASLR is likely to be unreliable in most scenarios.

On the other hand, we found that call sites for LoadLibrary and GetProcAddress functions were readily available within the Internet Explorer memory– 391 instances of GetProcAddress and 340 instances of LoadLibrary. During code page harvesting, we commonly find 10 or more instances of each function when starting from any initial address, and often find both within 100 pages. Note that the code page traversal strategy may be tuned (*e.g.*, depth vs. breadth-first ordering, direct vs. indirect disclosure ordering, etc.) to possibly find API references in fewer code pages on a per-exploit basis.

D. On Runtime Performance

Recall that every step of our framework occurs on-demand, at the time of exploitation. While we have demonstrated that we obtain enough code pages and gadgets under most circumstances, we have yet to discuss how long it takes before a payload may be successfully constructed.

To assess our runtime performance, we performed five end-to-end tests of the framework. The first scenario uses the proof of concept exploit against Internet Explorer 8 to launch an arbitrary sequence of commands on the victim's machine; similar to most existing proof of concept exploits, we chose to open the Windows calculator program. The second and third scenarios exploit a custom Internet Explorer 10 plugin on Windows 8. As no IE proof of concept exploits (stage one) have been publicly disclosed on this platform at the time of this writing, we embed both a straightforward ReadByte function into our plugin to disclose a byte of memory, as well as a buggy DebugLog function that contains a format string vulnerability. The code causing this vulnerability uses the secure family of printf functions[8], produces no compiler warnings in Visual Studio 2012, and is used by JIT-ROP to both read arbitrary memory via the %s modifier (see [49, Section 3.3.2]) and obtain the initial code pointer by reading up the stack to obtain the return address.

[8]For an overview of secure printf functions, see http://msdn.microsoft.com/en-US/library/ce3zzk1k(v=vs.80).aspx

Note that current prevention mechanisms implemented in Windows only protect against (uncontrolled) memory *writes* through a format string attack. The fourth scenario uses a rudimentary document reader program, called doxreader, that we created to support testing during development of our framework. The doxreader program contains embedded JavaScript support provided by Chrome's V8 engine. We also embedded a memory disclosure vulnerability within doxreader that we exploit to trigger code page harvesting. The fifth scenario demonstrates the native performance of JIT-ROP, which is run as a Linux executable (as described in §V-A).

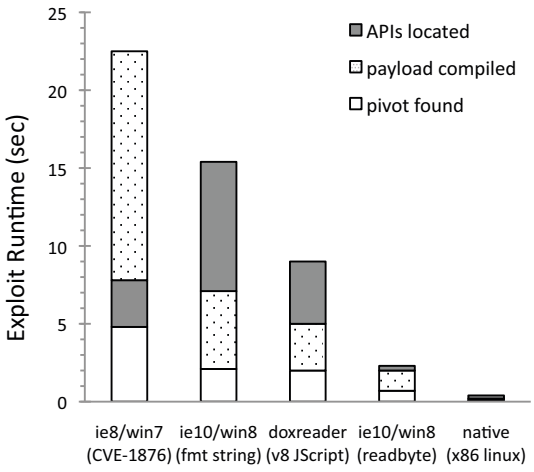

Figure 8. Overall runtime performance for end-to-end tests.

In our tests, the input to both IE and doxreader is a single JavaScript file that contains the entire framework and exploit-specific bootstrap to setup the memory disclosures. As in our offline tests, the input to the native executable is a memory snapshot of IE. For each end-to-end experiment we use JIT-ROP to produce a payload that launches the Windows calculator program, then exits the exploited process via the ExitProcess API call. The payloads generated are 100% ROP gadgets (*i.e.*, no injected code or secondary shellcode payloads) based on the memory content at time of exploit. Figure 8 depicts the results.

In our first scenario with Internet Explorer 8, code page harvesting was not very expeditious, averaging only 3.3 pages/second. Regardless, JIT-ROP was able to locate a pivot within 10 pages, all required APIs in 19 pages, and the requisite gadgets for a payload within 50 pages—a total running time of 22.5 seconds. In our second scenario (ie10/win8 (fmt string)), exercising a format string vulnerability to disclose memory (*e.g.*, swprintf_s is invoked for every two bytes disclosed) is a costly operation. In addition to requiring many more computations per byte disclosed, non-printable byte values cannot be disclosed at all, resulting in more pages being traversed because of

missed gadgets and API call sites. Regardless, we still see an increase in performance over the IE8 scenario with an overall runtime of 15.4 seconds and memory traversal averaging 22.4 pages/second, primarily because the JavaScript engine of IE10 uses JIT and typed arrays[9]. In comparison, when using the `ReadByte` disclosure in place of the format string vulnerability, we observe that the vast majority of overhead is caused by the type of memory disclosure itself. Under the `ie10/win8 (readbyte)` case, the exploit completes in only 2.3 seconds while traversing memory at an average of 84 pages/second. The `doxreader` exploit, using the V8 engine without typed arrays, completes in just under 10 seconds. Finally, notice that our framework runs incredibly fast when natively compiled—code pages are traversed, gadgets are collected, APIs are resolved, and a payload is compiled in a fraction of a second. While the exact performance of `JIT-ROP` varies significantly between JavaScript engine and exploit details, we believe the overall efficiency of our approach demonstrates the realism of the threat posed by our Just-in-Time code reuse framework.

VI. Related Work

As discussed earlier, exploit mitigation has a long and storied history (see [59] for an in-depth analysis). For brevity, we highlight the work most germane to the discussion at hand; specifically, we first review the fine-grained memory and code transformation techniques that have been proposed to date. In general, these techniques can be categorized into binary instrumentation-based or compiler-based approaches.

As the name suggests, binary instrumentation-based approaches operate directly on an application binary. In particular, Kil et al. [29] introduced an approach called address space layout permutation (ASLP), that performs function permutation without requiring access to source code. Their approach statically rewrites ELF executables to permute all functions and data objects of an application. Kil et al. [29] show how the Linux kernel can be instrumented to increase the entropy in the base address randomization of shared libraries, and discuss how re-randomization can be performed on each run of an application. However, a drawback of ASLP is that it requires relocation information, which is not available for all libraries. To address this limitation, several proposals have emerged [23, 45, 62]. Pappas et al. [45], for example, present an in-place binary code randomizer (ORP) that diversifies instructions within a basic block by reordering or replacing instructions and swapping registers.

In contrast, instruction location randomization (ILR) [23] randomizes the location of each instruction in the virtual address space, and the execution is guided by a so-called *fall-through map*. However, to realize this support, each application must be analyzed and re-assembled during a static analysis phase wherein the application is loaded in a virtual machine-like environment at runtime—resulting in high performance penalties that render the scheme impractical. Additionally, neither ORP nor ILR can randomize an application each time it runs. That limitation, however, is addressed by Wartell et al. [62], wherein a binary rewriting tool (called STIR) is used to perform permutation of basic blocks of a binary at runtime.

Recently, Giuffrida et al. [22] presented a fine-grained memory randomization scheme that is specifically tailored to randomize operating system kernels. The presented solution operates on the LLVM intermediate representation, and applies a number of randomization techniques. The authors present an ASLR solution that performs live re-randomization allowing a program module to be re-randomized after a specified time period. Unfortunately, re-randomization induces significant runtime overhead, *e.g.*, nearly 50% overhead when applied every second, and over 10% when applied every 5 seconds. A bigger issue, however, is that the approach of Giuffrida et al. [22] is best suited to microkernels, while most modern operating systems (Windows, Linux, Mac OSX) still follow a monolithic design.

With regard to compiler-based approaches, several researchers have extended the idea of software diversity first put forth by Cohen [15]. More recently, Franz [21] explored the feasibility of a compiler-based approach for large-scale software diversity in the mobile market. An obvious downside of these approaches is that compiler-based solutions typically require access to source code—which is rarely available in practice. Further, the majority of existing solutions randomize the code of an application only once, *i.e.*, once the application is installed it remains unchanged. Finally, Bhatkar et al. [7] present a randomization solution that operates on the source code, *i.e.*, they augment a program to re-randomize itself for each program run.

More distantly related is the concept of JIT-spraying [8, 47] which forces a JIT-compiler to allocate new executable memory pages with embedded code; a process which is easily detected by techniques such as JITDefender [14]. We also note that because scripting languages do not permit an adversary to directly program x86 shellcode, the attacker must carefully construct a script so that it contains useful ROP gadgets in the form of so-called unintended instruction sequences (*e.g.*, by using XOR operations [8]). In contrast, our technique does not suffer from such constraints, as we do not require the injection of our own ROP gadgets.

VII. Potential Mitigations

A knee-jerk reaction to mitigate the threat outlined in this paper is to simply re-randomize code pages at a high rate; doing so would render our attack ineffective as the disclosed pages might be re-randomized before the just-in-time payload executes. While this may indeed be one way forward, we expect that the re-randomization costs [62] would make

[9]For more information on typed arrays, see http://msdn.microsoft.com/en-us/library/ie/br212485(v=vs.94).aspx

such a solution impractical. In fact, re-randomization is yet to be shown as an effective mechanism for user applications.

Another JIT-ROP mitigation strategy could be to fortify defenses that hinder the first stage (*i.e.*, the entry point) of a runtime attack. For instance, the plethora of works that improve heap layouts (*e.g.*, by separating heap metadata from an application's heap data) [2, 5, 28], use sparse page layouts [38, 40] and heap padding [6], use advanced memory management techniques (*e.g.*, randomized [58] and type-safe memory re-cycling [2, 18]), heap canaries [46, 64], or a combination of these countermeasures in a single solution, which is exactly the approach taken by DieHard [5] and DieHarder [43]. Our proof of concept exploit (see §IV-E), for example, would be prevented by randomizing heap allocation order in such a way that heap Feng Shui is not possible. On the other hand, there are a number of other heap and information leakage vulnerabilities[10] that can be exploited to instantiate JIT-ROP and execute arbitrary malicious computations. Moreover, format string vulnerabilities, as demonstrated in §V-D, are a prominent class of attack that bypass these stage one defenses. A more in-depth overview of modern exploits that enable memory disclosures (in face of many of these defenses) is provided by Serna [51]. While stage one defenses certainly reduce the exposure of the initial vulnerable entry point, the functionality provided by JIT-ROP is orthogonal in that we bypass defenses against the execution of malicious computations (stage two).

Another potential mitigation technique is instruction set randomization (ISR) (*e.g.*, [4, 27]), which mitigates code injection attacks by encrypting the binary's code pages with a random key and decrypting them on-the-fly. Although ISR is a defense against code injection, it complicates code reuse attacks when it is combined with fine-grained ASLR. In particular, it can complicate the gadget discovery process (see §IV-C), because the entire memory content is encrypted. On the other hand, ISR has been shown to be vulnerable to key guessing attacks [57, 63]—that become more powerful in the face of memory disclosure attacks like ours,—suffers from high performance penalties [4], or requires hardware assistance that is not yet present in commodity systems [27].

Besides randomization-based solutions, a number of techniques have been proposed to mitigate the second stage of a runtime attack, namely the execution of malicious computations. However, most of these solutions have deficiencies which impede them from being deployed in practice. For instance, dynamic binary instrumentation-based tools used for taint analysis [42] or for the sake of preventing code reuse attacks [13, 17, 30] can impose slow downs of more than 2x. On the other hand, compiler-based approaches against return-oriented programming such as G-Free [44] or return-less kernels [34] induce low performance overhead, but require access to source code and a re-compilation phase.

A more promising approach to prevent control-flow attacks is the enforcement of control-flow integrity (CFI) [1]. CFI mitigates runtime attacks regardless of whether the program suffers from vulnerabilities. To do so, CFI generates the control-flow graph of an application and extends all indirect branch instructions with a control-flow check without requiring the application's source code. However, CFI still has practical constraints that must be addressed before it can gain widespread adoption. For instance, the original CFI proposal required debug symbols (which are not always available), and was based on an instrumentation framework ("Vulcan") that is not publicly available. Moreover, compiler-based follow-up works such as HyperSafe [61] or BGI [11] require a recompilation phase and the source code. But most importantly, prior work on CFI does not protect applications that deploy just-in-time code generation, which is the standard case for all modern browsers. Nevertheless, we hope our work encourages others to explore new ways to provide practical control and data-flow integrity.

VIII. Conclusion

Fine-grained randomization ([7, 23, 29, 45, 62]) has been recently introduced as a method of tackling the deficiencies of ASLR (*e.g.*, low entropy and susceptibility to information leakage attacks). In particular, today's fine-grained randomization defenses claim to efficiently mitigate code reuse attacks; a strategy used in nearly every modern exploit.

In this paper, we introduce a novel framework that undermines fine-grained randomization techniques by using a *just-in-time code reuse* strategy. Specifically, we exploit the ability to repeatedly abuse a memory disclosure to map an application's memory layout on-the-fly, dynamically discover API functions and gadgets, and JIT-compile a target program using those gadgets—all within a script environment at the time an exploit is launched.

In closing, one might question whether it is prudent to release details of a framework like ours given its ability to bypass both contemporary and next-generation defenses in a reliable manner. After much deliberation among the authors about ethical responsibilities, in the end, we believe that shedding light on fundamental weaknesses in current ways of thinking about exploit mitigation outweighs the potential downside of helping the next generation of hackers. If history serves any lesson it is that attackers will adapt and so must we in order to stay one step ahead. It is our hope that this work will inspire those more clever than ourselves to design more comprehensive defenses.

IX. Acknowledgments

The authors would like to thank Stefan Nürnberger, Teryl Taylor and Andrew White for fruitful discussions about this work. We also thank the anonymous reviewers for their

[10]For instance, CVE-2012-2418, CVE-2012-1876, CVE-2010-1117, CVE-2009-2501, CVE-2008-1442 to name a few.

insightful comments. This work is funded in part by the National Science Foundation under award number 1127361.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and Systems Security*, 13(1), Oct. 2009.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.

[4] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM Conf. on Computer and Communications Security*, 2003.

[5] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *ACM Conference on Prog. Lang. Design and Impl.*, 2006.

[6] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *USENIX Security Symposium*, 2003.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[8] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. In *Black Hat DC*, 2010.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symp. on Info., Computer and Communications Security*, 2011.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *ACM Conf. on Computer and Communications Security*, 2008.

[11] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles*, 2009.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conf. on Computer and Communications Security*, 2010.

[13] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*, 2009.

[14] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A defense against jit spraying attacks. In *IFIP International Information Security Conference*, 2011.

[15] F. B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12(6), 1993.

[16] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symp. on Info., Computer and Communications Security*, 2011.

[18] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, 2003.

[19] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, 1997.

[20] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *ACM Conf. on Computer and Communications Security*, 2008.

[21] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *New Security Paradigms Workshop*, 2010.

[22] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[23] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy*, 2012.

[24] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, 2009.

[25] V. Iozzo and C. Miller. Fun and games with Mac OS X and iPhone payloads. In *Black Hat Europe*, 2009.

[26] K. Johnson and M. Miller. Exploit mitigation improvements in Windows 8. In *Black Hat USA*, 2012.

[27] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Conf. on Computer and Communications Security*, 2003.

[28] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *ACM Conf. on Arch. Support for Prog. Languages and OSes*, 2006.

[29] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference*, 2006.

[30] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[31] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University, 2009.

[32] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf, 2005.

[33] H. Larry and F. Bastian. Andriod exploitation primers: lifting the veil on mobile offensive security (vol.1). Subreption LLC, Research and Development, 2012.

[34] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *European Conf. on Computer systems*, 2010.

[35] F. Lindner. Cisco IOS router exploitation. In *Black Hat USA*, 2009.

[36] D. Litchfield. Defeating the stack based buffer overflow exploitation prevention mechanism of microsoft windows 2003 server. In *Black Hat Asia*, 2003.

[37] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

[38] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ACM Conf. on Arch. Support for Prog. Languages and OSes*, 2008.

[39] D. Maynor. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress, 2007.

[40] O. Moerbeek. A new malloc(3) for openbsd. In *EuroBSDCon*, 2009.

[41] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), 2001.

[42] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security*, 2005.

[43] G. Novark and E. D. Berger. DieHarder: securing the heap. In *ACM Conf. on Computer and Communications Security*, 2010.

[44] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Annual Computer Security Applications Conference*, Dec. 2010.

[45] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[46] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *USENIX Conference on System Administration*, 2003.

[47] C. Rohlf and Y. Ivnitskiy. Attacking clientside JIT compilers. In *Black Hat USA*, 2011.

[48] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[49] Scut/team teso. Exploiting format string vulnerability. http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf, 2001.

[50] F. Serna. CVE-2012-0769, the case of the perfect info leak, 2012.

[51] F. J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[52] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communications Security*, 2007.

[53] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conf. on Computer and Communications Security*, 2004.

[54] Solar Designer. Return-to-libc attack. Bugtraq, 1997.

[55] A. Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.

[56] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista, 2008.

[57] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *USENIX Security Symposium*, 2005.

[58] C. Valasek. Windows 8 heap internals. In *Black Hat USA*, 2012.

[59] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Symposium on Recent Advances in Attacks and Defenses*, 2012.

[60] VUPEN Security. Advanced exploitation of internet explorer heap overflow (pwn2own 2012 exploit), 2012.

[61] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[62] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conf. on Computer and Communications Security*, 2012.

[63] Y. Weiss and E. G. Barrantes. Known/chosen key attacks against software instruction set randomization. In *Annual Computer Security Applications Conference*, 2006.

[64] Q. Zeng, D. Wu, and P. Liu. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *ACM Conference on Programming language design and implementation*, 2011.

[65] D. D. Zovi. Practical return-oriented programming. Invited Talk, RSA Conference, 2010.