2017 IEEE European Symposium on Security and Privacy

Revisiting Browser Security in the Modern Era: New Data-only Attacks and Defenses

Roman Rogowski,* Micah Morton,* Forrest Li,* Fabian Monrose,* Kevin Z. Snow,† Michalis Polychronakis‡

*University of North Carolina at Chapel Hill
{rogowski,micah,fcli,fabian}@cs.unc.edu

†Zeropoint Dynamics
kevin@zeropointdynamics.com

‡Stony Brook University
mikepo@cs.stonybrook.edu

***Abstract*—The continuous discovery of exploitable vulnerabilities in popular applications (*e.g.,* document viewers), along with their heightening protections against control flow hijacking, has opened the door to an often neglected attack strategy—namely, *data-only attacks*. In this paper, we demonstrate the practicality of the threat posed by data-only attacks that harness the power of memory disclosure vulnerabilities. To do so, we introduce *memory cartography*, a technique that simplifies the construction of data-only attacks in a reliable manner. Specifically, we show how an adversary can use a provided *memory mapping* primitive to navigate through process memory at runtime, and safely reach security-critical data that can then be modified at will. We demonstrate this capability by using our cross-platform memory cartography framework implementation to construct *data-only exploits* against Internet Explorer and Chrome. The outcome of these exploits ranges from simple HTTP cookie leakage, to the alteration of the same origin policy for targeted domains, which enables the cross-origin execution of arbitrary script code.**

The ease with which we can undermine the security of modern browsers stems from the fact that although isolation policies (such as the same origin policy) are enforced at the script level, these policies are *not* well reflected in the underlying sandbox process models used for compartmentalization. This gap exists because the complex demands of today's web functionality make the goal of enforcing the same origin policy through process isolation a difficult one to realize in practice, especially when backward compatibility is a priority (*e.g.,* for support of cross-origin IFRAMEs). While fixing the underlying problems likely requires a major refactoring of the security architecture of modern browsers (in the long term), we explore several defenses, including global variable randomization, that can limit the power of the attacks presented herein.

1. Introduction

Application exploitation has a long and storied history. No sooner are defenses deployed than attackers find ways to break down these barriers with ingenious tactics for injecting or executing arbitrary machine code in vulnerable applications. The *soupe du jour* relies on finding clever ways to chain together small instruction snippets, called *gadgets* [59], to implement arbitrary malicious logic—easily bypassing defenses such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR).

Admittedly, however, the exploitation of critical software, such as browsers and document viewers, is getting harder due to the deployment of additional security protections and exploit mitigations. Besides DEP and ASLR, sandboxing is increasingly used in applications that render untrusted input, control flow integrity (CFI) [2, 53, 68, 69] protections have also recently been integrated in some browsers, exploit mitigation toolkits such as Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [47] are being deployed in enterprise environments, and there have been many recent research efforts on additional protections, such as code diversification [7, 9, 31, 39, 52, 65] and execute-only memory [5, 15, 17, 28, 29, 30, 38, 45, 62, 66]. Undoubtedly, these new protections have improved the security landscape, in that although achieving reliable arbitrary code execution in the face of these protections may still be possible, they force the attacker to use more complex exploits that must chain together multiple vulnerabilities.

Given a memory-related vulnerability, however, control flow hijacking is not the only possible exploitation strategy. An arbitrary memory access or corruption capability can also be used to *read* private data, or *write* security-critical data. As we show later, the former can lead to the exfiltration of sensitive information through the available output channels of the vulnerable process, while the latter can be used to subvert security policies or protections and eventually lead to unauthorized data access or code execution.

Since the initial demonstration of such *non-control-data attacks* by Chen et al. [25], their full power seems to have gone unnoticed, at least until very recently [40, 41]. One possible reason may be rooted in the belief that "identifying security-critical non-control data and constructing corresponding attacks require sophisticated knowledge about program semantics" [25, p. 5]. Indeed, common wisdom is that the requirement of application-specific semantic knowledge and the problems presented by the limited lifetime of security-critical data are major factors that impose difficulties for attackers. While recent work has attempted to make it easier to identify and exploit security-sensitive information, *e.g.,* via coupling data-flow tracking, bug-finding tools, and semantic knowledge of the target program [40], we shed light on the fact that more practical strategies exist today.

DOI 10.1109/EuroSP.2017.39

To see why, consider the ubiquity of inbuilt scripting capabilities in complex, feature-rich applications, such as browsers and document-related software, along with their various plug-ins—all of which have been exploited because of memory corruption vulnerabilities. Recent work by Snow et al. [60] on *just-in-time* code-reuse attacks demonstrated how integrated scripting support provides powerful capabilities to adversaries, allowing them to dynamically interact with application memory by manipulating code pointers. Armed with a memory disclosure vulnerability, attackers can use embedded malicious scripts (*e.g.,* written in JavaScript or ActionScript), to access arbitrary memory locations in the address space of the vulnerable process, pinpoint useful gadgets, and synthesize them at runtime—thereby, effectively circumventing *code* diversification protections.

More worrying, however, is the fact that more and more of our online interactions involve web-based services, with the browser being the main gateway to them. Essentially, the browser is the new OS [64]. Thus, skilled adversaries will inevitability shift their focus away from needing to gain arbitrary code execution on the victim's machine, to instead leveraging *data leakage* attacks to find weaknesses in the Emperor's new armor. In doing so, attackers side-step the challenges mounted by the myriad of deployed protections against arbitrary code execution, and focus their attention on gaining security critical information that can be used to access the increasingly valuable amount of data stored in the cloud. For instance, stealing a user's session credentials for a cloud storage service may actually lead to even more data than what is available locally on a victim's device, let alone access to financial, e-commerce, and other online services.

We demonstrate the feasibility of constructing practical and reliable *data-only* exploits against modern browsers without deep semantic knowledge of the target programs. The main facilitator of our attacks is a technique we call *memory cartography*, which entails an off-line procedure that allows an adversary to construct a map of the target process' data objects and their references. Armed with an arbitrary memory read (or write) vulnerability and this precomputed memory map, an attacker can then use our framework in conjunction with malicious script code to *reliably* navigate through the process' memory *at runtime*, and reach critical data which can then be modified or exfiltrated.

2. Background and Related Work

2.1. Modern Browser Architectures

One contribution of this paper is bringing to the forefront a better awareness of significant security risks in modern browser architectures, especially those security practices that are undermined when the full power of memory disclosure attacks is realized. Thus, for pedagogical reasons, we first present a brief review of the state of the art in browser security. Interested readers are referred to previous works that explore the trade-offs between performance, compatibility, and security in the design and implementation of contemporary browsers over the past decade [6, 32, 56, 64].

For the most part, the evolution in the design of modern browsers relates to changes in the threat model they address. Early on, the perceived threat lied mainly in protecting users' systems from harm caused by malicious sites (*e.g.,* those that perpetrate code injection attacks to gain arbitrary control of the victim's machine), but over time, that model evolved to also attempt to provide isolation between distinct browser-based tasks, thereby not only protecting users but also the individual sites they visit from cross-origin attacks. Today's widely-used browsers have adopted a multi-process architecture that attempts to partition tasks into different processes, but the driving factor has been to improve responsiveness and robustness, rather than security [64].

Nevertheless, there are security benefits of moving towards a multi-process architecture. Given that the vast majority of bugs in a web browser exist in the code that parses or renders web content—due in part to the wide range of inputs that are processed by those routines, as well as their inherent complexity—a multi-process model offers a way to quarantine the rendering code to a process that does not crash the entire application if a bug in the rendering engine is exercised. In computer security parlance, the practice of relegating certain parts of an application to a lesser privileged process is commonly referred to as "sandboxing." Typically, the parsing and rendering of web content is relegated to a low-privileged sandbox process, so that in the event that security vulnerabilities in that code are exploited, the damage can be contained. For instance, in 2014 alone, *more than 600 security vulnerabilities* surfaced in Chrome's sandboxed code [27], but its multi-tiered architecture was touted for limiting the impact of these vulnerabilities.

The most common desktop browsers (*i.e.,* Mozilla Firefox, Internet Explorer, and Google Chrome) offer different levels of protection via their chosen sandbox mechanisms. To date, Chrome's designers have implemented what is considered to be the state of the art in browser security. Chrome is the only browser that completely restricts system-level access to the code that renders web content. Additionally, it offers additional security protections, including a limited support to render content from different domains in different processes, but such origin protection is only in effect when the experimental `--site-per-process` flag is used.[1] Such process isolation is implemented as a means to reinforce the same origin policy (SOP) [67].

The New Armor: Process Isolation. Given the strong security features of Chrome, we use it as a shining exemplar of a modern browser architecture. At a high level, the two main components of its sandbox architecture are the "broker" process (which runs at a regular user privilege level) and the various sandbox processes (which run with restricted system access). Different sandbox designs are used depending on whether they contain "renderer" code (*i.e.,* the

1. Firefox can isolate plugins to their own process, but features all of the web browser code in the same process [16]. Internet Explorer is somewhat of a hybrid between these two models—offering some sandbox-like protection schemes—but its architecture is not nearly as fully developed of a sandbox as Chrome [1, 57].

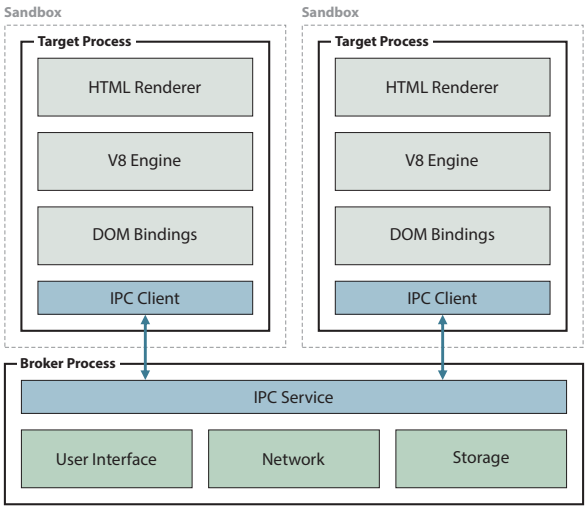

Figure 1: Chrome browser architecture.

part of the browser that parses and renders web content) or other components of the browser (*e.g.,* third party plugins). For brevity, we focus on the renderer sandbox, as this is where the core features of the browser reside. Examples of these core features include the DOM tree (including the browser logic that implements the same origin policy), the Javascript engine and the HTML renderer.

Figure 1 depicts the multi-process design of Chrome. The target processes at the top represent the sandbox instances, all of which communicate with, and rely on, the main broker process. To fully isolate damage that may be caused from activities within the target processes, Chrome's architecture further restricts the sandbox from gaining system access through direct calls to the operating system. To achieve such mediation, Chrome augments the sandbox process' security token in such a way that every request for system access is denied. Any file or network access that is needed by a renderer process to do its job is mediated by the broker process, which closely scrutinizes the renderer's request before carrying out the desired function. The access control enforcement mechanism implemented by the broker has at times contained flaws that allowed for sandbox privilege escalation [51], but the existence of this reference monitor offers yet another obstacle for an adversary who might be able to exploit a vulnerability in the renderer.

Hardening The Old Armor to Better Enforce SOP. Isolating the renderer is only one part of the protection offered by modern browsers. Isolating web content in one domain from web content in another domain by rendering each of these domains in different processes is another prudent security practice [26]. This practice follows from the realization that the same origin policy in browsers can be circumvented if an attacker can exploit a vulnerability in the renderer code. In essence, the same origin policy seeks to allow scripts from different web pages of the same origin (hostname, port, and protocol) to access all web content pertaining to that origin, regardless of which page actually rendered the content initially. On the other hand, a script from origin `a` is not able to access content belonging to origin `b`. In this way, if a web application opens a new window or tab in the same domain, both pages can freely access each other's content, but otherwise unrelated web pages are separated from each other. Thus, given two pages from origins `a` and `b`, the same origin policy attempts to prevent a number of misbehaviors, including (*i*) active content in `b` from successfully issuing arbitrary HTTP requests to `a`; (*ii*) scripts in `b` from accessing local or session storage that is meant to be specific to origin `a`, as it may contain sensitive information not meant to be shared across origins; (*iii*) scripts in `b` from accessing HTTP cookies that are specific to origin `a`, as these cookies are often used for authentication.

That said, the same origin policy is simply a logical framework that guides the interpretation of scripts and the rendering of content. Obviously, an adversary with direct access to a process' memory (*e.g.,* through a memory disclosure vulnerability) can conceivably undermine any protections provided by this policy. This issue was acknowledged as early as 2008 [56, 64], and is one factor that led to the rollout of Chrome's `Chromium-Sandbox` and other related app-isolation features [22]. The currently deployed `site-isolation` feature seeks to reinforce the same origin policy so that content from different domains is not only separated by barriers in the browser logic, but also by the operating system's ability to isolate processes from each other. Unfortunately, the complex demands of today's web functionality make the goal of enforcing the same origin policy through process isolation a difficult one to realize in practice, particularly when backward compatibility remains a priority. Our hope, in this paper, is to engage debate on whether the benefits of continuing along this path outweigh the risks that arise in the era of data-only attacks.

2.2. Exploit Mitigations and Data-oriented Attacks

Beyond the space of browser security, a wide array of preventative measures have been proposed to thwart both code injection and code reuse attacks—especially those based on return-oriented programming tactics [59] and related variants [11, 12, 14, 21]. These solutions attempt to either enforce some form of control-flow integrity (CFI) [2, 53, 68, 69] by following the principles first suggested by Adabi et al. [2], or diversify the code of the protected process to thwart the reuse of instruction sequences [7, 9, 31, 39, 52, 65]. More recently, a large body of work [5, 15, 17, 28, 29, 30, 38] has proposed the use of a combination of execute-only memory [45] coupled with randomized memory segments to undermine attacks that leverage memory disclosures as a precursor to building just-in-time code reuse payloads [60]. When source code is not available, alternative approaches [62, 66] enable binary compatibility by allowing the disclosure of code but prohibiting the execution of any code that was previously read (*i.e.,* via a memory disclosure).

The increasing complexity of reliably achieving arbitrary code execution due to the numerous deployed exploit

mitigations, along with recent incidents of data leakage vulnerabilities (such as Heartbleed [33]), has prompted a renewed interest into data-oriented attacks. Although the possibility of such attacks was previously known [25], data-oriented exploits against modern applications have only recently started to emerge [36, 48].

A first effort in systematizing the construction of data-oriented exploits was proposed by Hu et al. [40]. Their *data flow stitching* technique takes as input a vulnerable program with a memory error, an input that exploits that memory error, and a benign input that triggers the same execution path, and uses execution trace analysis (backward and forward slicing) to pinpoint data flow paths between inputs and pre-identified sensitive data. By automatically stitching different data flows, the resulting constructed exploit can either modify security-critical data to escalate privileges, or read and exfiltrate sensitive data. The goal of data flow stitching is different and complementary to this work, in that it performs heavyweight execution tracing of a program given a specific exploitable memory error, it does not work reliably in the face of ASLR, and it has been applied only on simple server programs.

In a follow up work, Hu et al. [41] demonstrated that data-oriented programming techniques can be used to undermine CFI defenses that trust the secrecy or integrity of security-critical meta-data in memory. That is, they show how turing-complete data-oriented attacks can be built that use data plane values for malicious purposes but maintain complete integrity of the control plane. Similar ideas in the control plane, that take advantage of flexibility in the intended code paths of real-world programs, were also demonstrated by Evans et al. [35]. Recently, Carlini et al. [19] apply generalizations of non-control-data attacks to show how an adversary can leverage a memory corruption to bypass the most restrictive CFI policies. As is the case with the techniques presented in this paper, CFI and related defenses offer little, if any, protection against our data-oriented attacks.

More recently, Jia et al. [42] showed that existing memory vulnerabilities in Chrome's renderer can be used as a stepping-stone to abuse the "web/local" boundary. Specifically, the authors show that because the security monitor's logic in Chrome is implemented as a set of function calls in the renderer module, data-oriented attacks can be used to undermine the SOP for scripts by changing the values of in-memory flags and data-fields for security checks. The security-critical data are identified via a best-effort approach that looks for differences in security-check functions across distinct execution traces recorded with the use of a debugger. In contrast, our goal is to show that we can *automatically map* the memory space of a process in a principled way, and enable *reliable* navigation through its data structures given an initial memory disclosure vulnerability, even in the presence of contemporary defenses like ASLR. Our focus is on modern web browsers—not just Chrome—but together, these works underscore the threat posed by data-oriented attacks that can completely undermine SOP enforcement.

Lastly, besides the enforcement of memory safety [3, 43, 50], other defenses against data-oriented attacks include data flow integrity [20] and data space randomization [8]. Privilege separation [18, 55] and hardware-enforced isolation of critical data such as cryptographic keys [49, 63] can also mitigate the threat of data leakage attacks. We return to these and other defenses later on in §7.

3. Adversarial Model

Throughout this work, we assume that the following widely-accepted protections are enabled:

1) Data Execution Prevention
2) Address Space Layout Randomization
3) Modern Protections in Windows, including the extensions available in the Enhanced Mitigation Experience Toolkit (EMET) [47]

With regards to our adversarial assumptions, we assume that adversaries are aware of a memory disclosure vulnerability that allows them to read and write arbitrary memory locations. As several recent works [10, 24, 28, 37, 46, 58, 60] have shown, these assumptions are *no stronger* than the capabilities leveraged by skilled adversaries to defeat contemporary ASLR. Additionally, we assume that *state-of-the-art sandbox protections are deployed* within the parent process of the victim's browser or application. We also remind the astute reader that even fine-grained ASLR schemes (of which many have been proposed in the academic literature over the past few years [4, 15, 30, 38, 52, 65]) might be assumed here; unfortunately, since all such schemes we are aware of leave `.data` sections untouched, they too offer little protection against the techniques discussed next.

4. Memory Cartography

A key observation in this paper is that although security and privacy relevant data is well known to be present in application memory (which is *non-contiguous*), techniques for reliably bridging these isolated memory regions to navigate memory are not immediately obvious. Indeed, one challenge when leveraging a memory disclosure vulnerability lies in navigating from a *source* memory region (*i.e.,* the initial disclosure point) to a *destination* memory region (*i.e.,* where the sensitive data resides), while ensuring that invalid memory addresses are not dereferenced [37, 60], as doing so will crash the target process. Crashing the application would spell disaster for the adversary, as the application would quit before exploitation could be carried out.

As we began our deep exploration into data-only attacks, however, we realized that there are application and operating system agnostic techniques that allow one to navigate non-contiguous memory without following any code pointers (which would be thwarted by current code randomization defenses [28, 29]). Sadly, the discovered techniques aptly demonstrate the power (and ease) of data-only attacks—whether ones that merely disclose information to the adversary, or ones that completely control a victim's browsing session by subverting the same origin policy. We call our

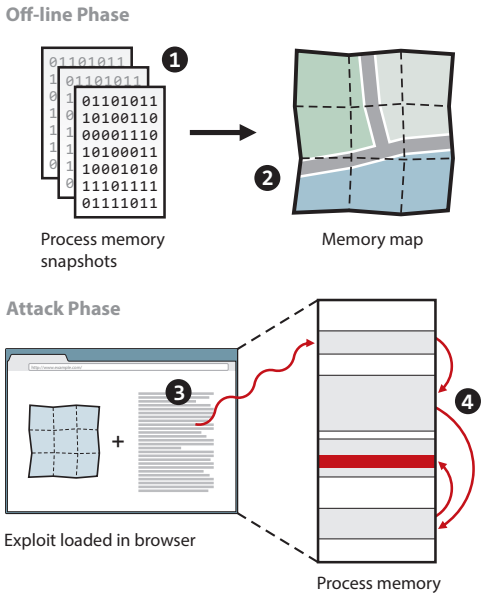

Figure 2: Workflow using the `Pathfinder` framework.

techniques for navigating non-contiguous memory regions *memory cartography*.

To show that these memory cartography techniques are effective across varying operating systems and applications, even in face of widely deployed defenses, we designed and built a prototype framework, called `Pathfinder`, which demonstrates one instantiation of the concept. The overall workflow (shown in Figure 2) is as follows: First, an adversary uses `Pathfinder` off-line to harvest in-memory *data pointers* of the target process (Step ❶) and automatically construct a portable memory map prior to exploitation (Step ❷). Armed with a memory disclosure vulnerability adapted to conform to a simple interface implementing `DiscloseByte` and `WriteByte` functionality, the attacker then provides an initial code pointer (Step ❸). With this as a starting point, the attacker can then use data-only memory manipulation scripting primitives to safely navigate through the address space by dereferencing further pointers according to the precomputed map, and disclose or rewrite critical portions of memory (Step ❹) without the risk of accessing an invalid memory address and crashing the application.

The implementation was highly involved, and successful completion had to overcome several challenges. Nevertheless, we show in §6 that our implementation of `Pathfinder` 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 necessity of memory cartography, the intuition of our approach, and necessary components of our system. We later detail a concrete example of an exploit on Chrome's sandboxed renderers using our framework in §6.

4.1. Navigating Memory

The overarching goal in this section is to disclose or overwrite a *destination* memory region (*e.g.,* on the process stack, heap, or the "global" variables located in a code module's `.data` region) that contains sensitive data, by leveraging an arbitrary memory disclosure vulnerability. For example, one may wish to discover the value of an HTTP cookie to hijack a victim's authenticated session on another domain. Through manual program analysis, we know that modern browsers store cookies (at least transitionally) on one of several process heaps. In this instance, consider a scenario wherein the adversary has exploited an ActionScript (*i.e.,* Adobe Flash) vulnerability to overwrite the *length* field of a string stored on the Flash heap with the maximum integer value. By indexing into that string, the adversary can effectively access arbitrary addresses relative to that string. To enable arbitrary *absolute* reads or writes, that relative read can be used to disclose a pointer reference to the string (or any other object adjacent to it) *in the same heap region*, which does not require navigating between isolated memory regions. Once the address of the exploited string is known, relative accesses can be adjusted to absolute accesses by taking the difference of the desired address and the exploited string object's address.

At this point, although the adversary can read or write to arbitrary locations in the vulnerable process' address space, locating the actual cookie remains a challenge due to the *fragmented* nature of the address space. In particular, although the cookie (*destination*) resides in the heap used by the browser's DOM rendering code, the only known valid address (from the attacker's perspective) is that of the manipulated string, which resides on a *different* heap (the Flash heap). The adversary has no method of bridging this disconnect. Were one to simply guess program addresses for the destination, the application would immediately crash on an incorrect guess, and hence the exploit would fail. Ideally, the vulnerable application would instead have exception handlers in place around the code used to dereference memory. Gawlik et al. [37] have taken advantage of this particular scenario, but we instead seek a generic, application and operating system agnostic approach to safely reach critical objects in a process' address space. Our approach for solving this problem in the general case is rooted in the inherent presence of runtime linkages (pointers) between different memory regions.

4.2. Exploration of Runtime Data Pointers

We start with the hypothesis that distinct regions of memory, while not contiguous, are in fact frequently linked by pointer references between them at runtime. If this hypothesis holds, and the memory regions are well-connected, one should be able to follow these linkages to navigate from a source region, to zero or more intermediary regions, and then to the destination region. Hence, the question arises of whether these linkages are present, whether they are strongly connected, and if they can be reliably identified

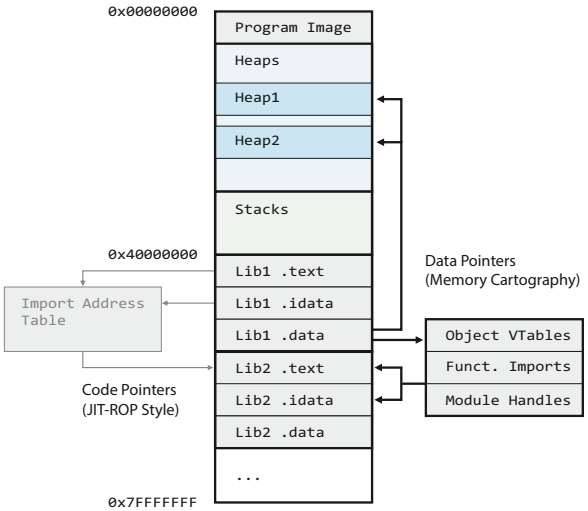

Figure 3: Example process memory address space.

in face of address space layout randomization. The work of Snow et al. [60] demonstrated that *code linkages* can be reliably followed in face of fine-grained ASLR. In this work, however, we are not concerned with disclosing gadgets to use in a code reuse attack, but rather with reaching a specified *data destination* (*e.g.,* executable or library globals, stacks, and heaps). While the aforementioned execute-only memory primitives aim to mitigate code linkage disclosures used in [60], our approach sidesteps that mitigation, as data sections must remain readable.

For the remainder of this paper, we take a conservative approach and consider a *data pointer* as simply a pointer in the `.data` section of an executable or (shared) library that points to any other valid memory location, be it a memory region that represents code, data, or a hybrid of both code and data, as illustrated in Figure 3. Let us consider a few scenarios that support our hypothesis of data-pointer linkages. Regarding data pointers to heaps (or fragments of heaps), a globally defined *object reference* (in C++ parlance) will, at some point during the execution of an application, be initialized via a *malloc* or *new* instantiation, which allocates the data for that object on one of the program heaps. In turn, that global object reference will be set to point to that location within the heap. Hence, data pointers (in the `.data` section) should exist at that point to program heaps. Similarly, objects may be allocated on the stack (*i.e.,* function-local objects), then later assigned to a global reference (more on this in §6). Hence, data pointers to stacks should also exist in `.data` sections.

Perhaps more important is the existence of data pointers to *other libraries*, which themselves will contain more data pointers to other heaps and stacks. For example, consider a function that dynamically links to a shared library at runtime (*e.g.,* using `LoadLibrary` in Windows or `dlopen` in Linux or OSX). Such a function would need to update a dynamic linker structure in memory. This structure is responsible for maintaining a list of all shared libraries mapped to process

Algorithm 1 `HarvestDataPointers`: discover runtime data pointers to other libraries, stacks, and heaps.

Input: M {set of valid program memory regions}, D {set of program `.data` regions}, P {size of pointer in bytes}
Output: E {set of edges between memory regions}
for $region \in D$ **do**
  for $i = 0; i < |region|; i$ += P **do**
    $ptr = region_{i:i+P}$
    if $\exists(ptr \in M)$ {valid address} **then**
      {store valid edge}
      $E(region_{name}, region_{base} + i) \leftarrow (M(ptr)_{name}, ptr)$
    end if
  end for
end for

memory, and is located in user space for the Windows, Unix, and Mac OS X operating systems. As such, one would expect that pointers to these structures would reside somewhere in the libraries implementing such functionality.

Indeed, our initial investigation revealed a multitude of inter-region data references in several popular applications. We detail the number of references and their types later, but at this point it is sufficient to mention that these references come from a variety of data sources pervasive in common coding constructs, such as virtual function tables (VTables), function pointers, library base address references, imports, and variable definitions of objects allocated on the stacks and heaps. With the intuition of why data pointer linkages exist, the next section details how to leverage this information to construct and use a portable memory map.

4.3. Our Approach

4.3.1. Step ❶: Data-Pointer Harvesting (Off-line). Under the assumption that such data linkages exist, our approach for navigating from a *source* memory region to a *destination*, without dereferencing an invalid address, is to construct a map consisting of edges from a source data region (the `.data` section of an executable or library) to a destination code (*i.e.,* an executable or library `.text` section) or data (a heap, stack, or another `.data` section) region. As our goal is to achieve this navigation in an application and operating system agnostic way, we approach the problem with the assumption that no semantic knowledge about any data pointer is available. Further, recall that linkages will not statically exist in the application executable and library files because all these pointers are dynamically determined at runtime. That is, the `.data` sections will hold null references both in executable files and memory until the process initializes those values at various points during execution.

Hence, we start by launching our target application and performing the basic program actions (manually) that a victim might take. For instance, for web browsers, we navigate to several web pages, or for document readers, we simply open a document. In performing these actions, libraries are loaded and initialized, along with the variables in their data sections. At this point, we pause and examine

the process memory to construct a memory map for this particular instance. One way to construct such a memory map is to scan the data sections of all modules (*i.e.,* the main executable and shared libraries) and interpret all 4-byte (32-bit) or 8-byte (64-bit) aligned values as pointers. Whenever such a data pointer to a valid memory location is found, an edge is created between the *source* and its corresponding *destination* location. This process is detailed in Algorithm 1. The idea is that enough edges will be generated to fully connect a graph of all mapped data memory regions.

Of course, the memory layout of an application running on one system will likely be different from an instantiation on another system due to executable and library *relocations* wherein entire modules are loaded at different base addresses. This can occur by coincidence (for compatibility reasons) or as a result of address-space layout randomization. In either case, the map would be useless without making it portable across differently randomized runs. Next, we discuss how we address this challenge.

4.3.2. Step ❷: Generating a Portable Memory Map. The edges identified in §4.3.1 suffer from two problems: (*i*) using absolute addresses for source and destination locations is not portable across randomized program instances, and (*ii*) we have no assurance that the identified links are not false positives, *i.e.,* data values that happen to reference valid memory locations when they are interpreted as pointers, but which actually are just arbitrary data, and thus not consistent across different runs of the same application. Without resolving these issues, an exploit following the links in our map will likely dereference an invalid address and crash the program before completing its malicious actions.

The first problem is resolved by storing links as relative offsets into a specific library. Although ASLR relocates libraries, relative locations *within* a library are not randomized. Hence, vertices in our memory map graph take the form of *(src_name, src_offset)* $\rightarrow$ *(dst_name, dst_offset)* where *name* is a designation of a library's file name (*e.g., libc*, *kernel32*), a specific thread's stack (*e.g., stack1*, *stack2*), or a process heap (*e.g., heap1*, *heap2*). The implication is that a single leaked data pointer to a memory region will be enough to determine the exact location of the data section of the module in which that region resides.

To reduce false positives, we apply a rather simple, but effective, approach: we simply repeat the above procedure after rebooting the machine on which the off-line map is being generated. By rebooting, the effects of ASLR (wherein the location of all libraries in program memory are re-randomized) can be effectively dealt with. That is, some pointers that by chance dereferenced to valid memory locations will now point to invalid memory, or perhaps to a different memory region altogether. By only keeping data pointers that *consistently* point to the same offset in the same memory region, we are able to eliminate all false positives after a few iterations of refinement (see §5 for more details).

At this point we have a (hopefully) well-connected graph of memory region linkages. As an optimization, we can remove duplicate edges (*e.g.,* duplicate links from one region to another), to consolidate the graph. Next, we show how this memory map can be used.

4.3.3. Step ❸: Adapting a Memory Disclosure for Memory Cartography. With a memory map in hand, the next step is to adapt a memory disclosure to our interface, and leverage the framework to access or modify any sensitive data indexed by the map. To do so, a `DiscloseByte` function must be implemented on a per-exploit basis that reveals the value of a single byte of memory at a given address. If the goal is simply to disclose sensitive information, as in our cookie example in §4.1, then this one function is all that is required. On the other-hand, if the goal is to overwrite arbitrary values in memory, then the memory disclosure must also be adapted to *write* arbitrary values of memory with a `WriteByte` function.

In many cases involving memory disclosures enabled by script-accessible vulnerabilities, the adaptation is straightforward—in our string exploit example in §4.1 the solution is obvious: just modify the string instead of reading it. In other cases, as for example with the *HeartBleed* memory disclosure vulnerability, it is not possible to write memory. While this capability depends on the circumstances of the exploit, it is safe to assume that script-oriented exploits frequently offer this power to the adversary, and so we explore the case of an adversary who has this ability. Built atop these two primitives, we provide the following primitives:

ReadMemory · WriteMemory Reads or writes an arbitrarily sized chunk of memory with repeated uses of `DiscloseByte · WriteByte`.

ShortestPath Given a *source* memory region name (*e.g., jscript9*), an address and *destination* memory region name (*e.g., heap2*), uses the memory map to return Dijkstra's *ShortestPath* to the destination.

MemoryRegion Given a source memory region name and address, as well as a destination memory region name, dereferences the chain of links in the *ShortestPath* to return the destination region's start address and size.

Stacks · Heaps · Libraries Given a source memory region name and address, uses *ShortestPath* and *MemoryRegion* to return a list of addresses to all known stacks, heaps, or libraries.

HeapBase Given a pointer to a heap, finds the base address of the heap by stepping backwards through memory until the OS-specific heap header is found.

FindMemory · ReplaceMemory Given a *MemoryRegion*, returns the chunk(s) of memory that match the specified regular expression or replaces it with the specific buffer of data.

SnapShot Given a source memory region name and ad-

dress, finds a path to every other memory region and returns an application memory snapshot (*e.g.,* `core`, `minidump`).

By now the astute reader should recognize the unbridled power of combining a memory map with a memory disclosure: the adversary now completely controls the application's data from a script without any risk of crashing the process due to an invalid address dereference. Next, we first briefly discuss how to leverage this power, and later provide a concrete example in §6.

4.3.4. Step ❹: Following Data Pointers at Runtime. Given the memory map constructed in Steps ❶ and ❷, the adversary's exploit-specific implementation of `DiscloseByte` and `WriteByte`, and the helper routines provided by our framework in Step ❸, let us now return to the cookie disclosure example put forth in §4.1. We left off at a point where the adversary obtained a pointer to the Flash heap (where the exploit took place), but ultimately needs to navigate to the heap used by the browser's DOM rendering code. Recall that we only inspect the `.data` sections of executable and library regions for edge sources, while heaps and stacks are only considered as destinations. This is intended, as the relative offsets of our pointers within a heap will likely change from one application run to another. Hence, the adversary must provide an initial source memory region that represents either the main executable module or any of the loaded libraries.

Identifying the address of a library can be done given that one can already read arbitrary data on the initial heap. At a high-level, the memory disclosure vulnerability can be used to leak a VTable pointer of any heap object, and using in-built scripting (*e.g.,* Flash) one can instantiate any number of these objects on the heap. The VTable will contain pointers to the machine code implementing the object's methods, which means that the adversary can first disclose a VTable pointer, then reach a code pointer pointing to the library implementing the object's function within that table. Even in face of ASLR, this function pointer will be at a fixed offset from the library's base address.

With a source library in hand, the adversary can follow the shortest path to the heap used by the browser's DOM rendering code to arrive at the given destination, dereferencing each pointer along the way to arrive at the next. Once the adversary has navigated to their destination, it is simply a matter of searching that chunk of memory to find the relevant information to disclose. At this point the possibilities are endless. We explore some possibilities later in §6 after detailing our cross-platform implementation.

5. Implementation

To evaluate our memory cartography technique's applicability to a wide variety of applications and operating systems, we developed a cross-platform implementation of our `Pathfinder` prototype framework on 32-bit Microsoft Windows, 64-bit Apple OSX, and 64-bit Linux in C++. Most of the framework code is application-independent, *e.g.,* the code that implements data pointer scanning, graph construction and pruning, and shortest path computation, but each OS required a small amount of platform-specific code to (i) obtain a ground truth memory mapping for each application, as reported by the OS, and (ii) read the contents of the target process' memory. For obtaining the true memory map during our off-line data pointer gathering phase (Step ❶) we used the `NtQueryVirtualMemory` API on Windows, the proc filesystem's `/proc/[pid]/maps` file on Linux, and the `vm_region` API on OSX. To read a remote application's memory during map construction we used `ReadProcessMemory`.

As discussed in §4, the off-line stages of the memory cartography step require that we take an iterative approach in order to eliminate false positives. Table 1 reports the number of false positives eliminated from the memory map in each pass after the map's initial construction. Notice that by the second iteration, only a few false positives remain, and for the applications we tested, false positives were completely eliminated within three iterations.

TABLE 1: False Positives Remaining (Memory Map)

Application	1st Pass	2nd Pass	3rd Pass
Internet Explorer	26326	1	0
Google Chrome	1549	0	0
Firefox	13071	20	0
Adobe Reader	78231	4	0

To provide the reader with more insights into how connected the nodes are in the maps we generate, Table 2 shows the *edge connectivity* of each memory map. Edge connectivity indicates the minimum number of edges (see 4.3.1) whose removal disconnects all paths between sources s and sinks t of interest in the proof of concept. In this case, all edges are assigned a weight of 1.

We also explored the *minimum s-t cut* for each graph, wherein *system* libraries (*e.g.,* any shared library that is not specific to an application) are assigned a weight that is one half the cost of non-system libraries. We performed this analysis as one might argue that it is more natural to expect system shared libraries to exist on a similar, but not necessarily identical, target machine where the map is going to be used at runtime. Hence, we are interested in the minimum cut that optimizes paths through system libraries. Intuitively, the closer the ratio between edge connectivity and the minimum *s-t* cut is to 1, the more portable the map.

Table 2 shows the results prior to the refinement of removing duplicate links to consolidate the graph. Note that for IE, Adobe Reader, and Firefox, we observe that the minimum *s-t* cut is close to the edge connectivity, suggesting that maps are robust. Interestingly, for Chrome, the ratio is larger, but we note that this is because the average length of the path from s (`chromechild.dll`) to t (a custom heap) is 1, and that particular sink is a non-system DLL.

While Table 2 exemplifies the overall connectivity of our resulting map, navigating to some regions is more useful than others. For instance, many of the concrete attacks we enumerate in Section 6 require access to the program heap. It

TABLE 2: Memory Map Statistics

Application	Source (s)	Sink (t)	Min ; Avg Path	Connectivity	Min s t Cut
Firefox	`xul.dll`	`ntdll.dll`	1 ; 7	80	87
Adobe Reader	`npswf32.dll`	`ntdll.dll`	1 ; 8	41	43
Internet Explorer	`Flash.ocx`	`ntdll.dll`	1 ; 3	11	11
Google Chrome	`chrome_child.dll`	Custom Heap	1 ; 1	4	8

is worth noting that we identify linkages to at least one segment of each application heap in all experiments. However, the pointers outgoing from those segments will vary between runs, as object addresses allocated on the heap are not deterministic. As such, with heaps we cannot use the same general strategy of following pointers from a fixed offset of the incoming edge, as those offsets vary. Hence, heap segments in our map currently only have incoming edges. Regardless, the segments we identify using Algorithm 1. alone successfully facilitate the attacks outlined in section 6. If necessary, however, one could use heap layout-specific knowledge to parse and enumerate all heap segments and further increase map connectivity.

The construction of a memory map for a given target is a quick process. Indicatively, constructing the memory map for `chrome.exe` of Google Chrome—which requires parsing several hundred MB of memory—takes under a minute, while the same process for `iexplore.exe` of Internet Explorer is even faster, in the order of a few seconds.

5.1. A Note on Cross-Library Data Pointers

With an implementation at hand, we ventured to better understand *why* there are data pointers that reference libraries. In the remaining discussion, we focus on Internet Explorer 11, since 98% of all data pointers in its memory map had associated debug symbols, and the existence of these symbols helped our understanding of these linkages.

Of all the data pointers that had associated debug symbols, 81.2% of them were referencing VTables in other shared libraries. This seems to be a result of several practices, including using an Object-Oriented Programming paradigm. For ease of exposition, suppose that class `F` is implemented in shared library *LibA*. Class `F` likely consists of virtual methods, which are inherited by all objects instantiating `F`. Now, suppose an object `f_obj` of class `F` is created in a different shared library, denoted *LibB*. The code that implements `f_obj`'s virtual functions resides elsewhere, so there will be a linkage in `f_obj`'s VTable referencing the code in *LibA*.

In addition to VTable pointers, we observed two other main types of data pointers: `__hmod__` and `__imp__`. The former are references to the base addresses of other shared libraries, while the latter are simply function imports. Although these function pointers exist outside of the `.idata` sections of shared libraries (which typically contain function pointers for imports), we observed that `__imp__` pointers are imports not included in the `.idata` sections. We hypothesize that these function pointers are global variables instantiated using the `GetProcAddress` Windows API call. We return to a discussion on disallowing cross-compartment pointers as a mitigation strategy in §7.

6. Evaluation

We now return to our discussion on the same origin policy (SOP) in modern browsers (§2) to demonstrate the power of our memory cartography technique by shedding light on its real-world implications. We first elaborate on a common overarching scenario that facilitates the attack, then detail two (of the hundreds of disclosed vulnerabilities) that could be used to disclose memory in a browser's renderer process, and finally present real-world examples of SOP-violating actions. These examples highlight the urgency to address weaknesses in the current thinking about content isolation and browser security given the ease with which one can take advantage of data-only attacks.

6.1. Example Scenario

For pedagogical reasons, our attack scenario begins like any other so-called *drive-by download*. That is, we assume that the adversary has compromised some vulnerable web server and has embedded a hidden IFRAME that secretly directs all subsequent visitors of that site to the exploit landing page. The goal, however, is *not to inject code into the browser and ultimately install malware—we assume that the browser's renderer is sandboxed, and that the sandbox does not allow the browser code, whether newly injected or not, to interact with the underlying operating system in any meaningful way*. Furthermore, we assume that the broker process, which acts as a gatekeeper between the sandbox and the operating system, does not not contain any logical flaws or memory error vulnerabilities, nor do the OS-level system calls. Hence, we are *isolated in the renderer process*.

It is expected that many of the victims interact with a plethora of other web sites prior to reaching the adversary's page, whether in that same browsing session (hence, likely using *session cookies*) or at some point in the past (where *persistent cookies* are used). Our initial goal is to obtain the cookies from the victim's browser. As cookies are often used for authentication, the adversary can use those cookies to access the victim's emails, shopping and credit card information, home address and phone number, banking account numbers, and even profile their personal (and possibly embarrassing) habits.

In the past, attackers followed one of two routes to obtain this information. They either exploit a web site-specific *cross-site scripting* (XSS) vulnerability, which is quite limiting considering that only the one web site's

credentials can be leaked, or the adversary compromises the entire system via code injection or reuse to install malware. Once the malware was installed, one could exfiltrate not only cookies, but also documents on the local file system. Sandboxing, however, has severely handicapped those attacks in recent years. Data-only attacks are a middle-ground, both for attackers and defenders, in that the sandbox quarantines the spread of injected code, but as we will demonstrate adversaries still have at their fingertips a capability that far exceeds any individual XSS, *cross-site request forgery* (CSRF), "clickjacking," or any other attack at the web application layer. Indeed, an attacker using our framework holds a capability that exceeds holding any web-based vulnerability for all web sites combined.

To reap the benefits of our framework in a sandboxed web browser, we need to first (i) render each domain of interest *in the same process* as our attack domain, which we call *process feng shui*,[2] so that a remote domain's application memory is available to us, then (ii) exploit any memory disclosure vulnerability to conform to our `DiscloseByte` · `WriteByte` interface, so we can read and write that memory at will from our script, and finally (iii) navigate memory with our data-only primitives to grab cross-domain information and overwrite security-relevant variables in memory to violate cross-origin policies where needed.

6.1.1. On finding target variables. We note that finding security-relevant variables in memory is a non-trivial process, but it is still easier than one might expect. For instance, Chrome has a *code search feature* which can be used to find variables of interest, *e.g.,* cookies. When public debug symbols are available, they can also be used to help identify security-relevant variables. Moreover, in the case where the target application offers security-related settings or flags (*e.g.,* enabling Flash, disabling SOP, or disabling warnings), a rather effective technique is to perform data-flow tracking (*e.g.,* using popular tools [44]). For the attacks in this paper, we leveraged both code search and data-flow tracking facilities to quickly identify targets. Skilled adversaries would do the same.

6.1.2. Process Feng Shui. A precondition for the application of our memory mapping attack to web browsers is the ability to cause web content from different domains to be rendered in the same process. Unfortunately, under the current process isolation model of Chrome, newly created tabs as well as cross-origin browser-initiated navigations (*e.g.,* address bar or bookmarks) will generally cause a process swap, while renderer-initiated navigations (*e.g.,* links, IFRAMEs, `window.open()`) stay in the same process. This means that there are many cases of content from different domains being rendered in the same process. An attacker can, for example, programmatically achieve this by either including content from a different domain in an IFRAME, or automatically opening a new page from a different domain using an `A` tag or by calling the `window.open()` JavaScript function, an approach that we dub *process feng shui*.

```
var vuln = null;
var vuln_addr = null;

function InitExploit() {
  var a = new ArrayBuffer(SMALL_BUCKET);
  a.__defineGetter__('byteLength', fuction() {
    return 0xFFFFFFFC;
  });
  vuln = new Uint8Array(a);
  vuln_addr = vuln[offset_to_ptr] -
    offset_from_vuln;
}

function DiscloseByte(addr) {
  if (vuln_addr > addr)
    return vuln[0x10000000
              + (addr-vuln_addr);
  else return vuln[addr-vuln_addr];
}

function WriteByte(addr, byte) {
  if (vuln_addr > addr)
    vuln[0x10000000
        + (addr-vuln_addr)] = byte;
  else vuln[addr-vuln_addr] = byte;
}
```

Listing 1: Chrome CVE-2014-1705 implementation of `DiscloseByte` · `WriteByte`.

Chrome's security architecture does not include a way to render these different domains in different processes without breaking compatibility for cross-origin IFRAMEs. That is, if a malicious site embeds an IFRAME to google.com, the rendering engine loads the IFRAME in the same process as the parent page, thus making the cookies and other origin state accessible to the (possibly exploited) process. This is true of all major browsers today. Thus, an attacker can use one of the above two methods to trigger the rendering of web content from a target domain in the same context as the malicious page's domain, and subsequently subvert the same origin policy to steal private data from the victim domain—all within the confinement of the sandbox.

For completeness, we briefly address how an attacker may perform similar process feng shui in the other two browsers considered in this work (*i.e.,* Firefox and IE). Note that we assume the attacker can trigger a vulnerability in the browser's rendering code itself, rather than exploiting third party plugins, which are at times rendered in different processes than the browser. When mounting this attack against Firefox, the attacker does not have to perform any *process feng shui* at all, since all rendering code in Firefox takes place in the same process.

Internet Explorer, on the other hand, does feature a multi-process architecture that requires some manipulation by the attacker. Sadly, IE only places the rendering of different web pages into different processes for the sake of performance and reliability, rather than reinforcement of the same origin policy. Thus, the multi-process architecture of IE can be overcome through *process feng shui* on the part

2. A play on *heap feng shui*, which is used to precisely arrange objects on a process heap prior to exploitation.

of the attacker, due to the following design choices in IE: (1) Tabs are only rendered in a new separate process when there is sufficient system memory. The attacker's page can allocate a significant fraction of system memory, thus ensuring that IE will decide to place the victim tab in the same process as the malicious page. (2) New tabs are assigned to IE processes in a predictable round-robin fashion. Thus, by using varying degrees of *process feng shui*, in all browsers we tested an attacker can always reliably cause content from an arbitrary domain to be rendered in the same process as the malicious domain.

6.1.3. Memory Disclosure. At this stage, we assume that the adversary has pinpointed information of interest into the browser's memory, but still needs to leverage the `DiscloseByte · WriteByte` interfaces to utilize the primitives provided by the framework to subsequently disclose the desired information. To exemplify this process, we demonstrate how one can implement the primitives in both Internet Explorer 11 via the Flash vulnerability CVE-2015-0359, and in Chrome 33 via the JavaScript vulnerability CVE-2014-1705.[3]

The Chrome vulnerability takes advantage of the *defineGetter* feature, which enables one to (re)define properties of JavaScript objects. However, some key properties should not be re-definable, such as an array length, which is exactly what this exploit takes advantage of in Listing 1. To setup the exploit, a new array is created in line 5, then its length property is redefined to always indicate a length several bytes short of the largest 32-bit integer in lines 6–8, and a new byte array is defined in line 9 that uses that new definition. Given that the adversary can now index into any offset of an array, regardless of its actual allocated size, any address relative to the start of that array can now be read.[4]

Finally, we use *heap feng shui* to arrange objects on the heap nearby the array (not shown in listing) that contain self-references, then use the exploited array to perform a relative read and disclose one of these references and determine the absolute address of the array in lines 9–10. At this point, `DiscloseByte · WriteByte` interfaces are implemented by adjusting the given absolute address to the corresponding address relative to the exploited array and reading or writing at that offset.

The Flash vulnerability used for Internet Explorer, however, is more complex in its implementation. In short, CVE-2015-0359 is a *use-after-free* vulnerability wherein an ActionScript worker attempts to use a ByteArray (the *domainMemory* member variable of the *flash.system.ApplicationDomain* class) after it was already freed by another worker. To implement the `DiscloseByte · WriteByte` interfaces, we again use *heap feng shui* to ensure that another object is allocated in the freed *domainMemory* slot before it is reused. We then use this newly allocated object to modify the bytes in that memory slot that correspond to the offsets where the expected ByteArray object's *length* member variable is stored. As a result, the new worker's *domainMemory* can now access data at any offset similar to the replaced array length in the Chrome exploit. Hence, from this point on the primitives are implemented identically to those in Listing 1.

We also note that *heap overflow* vulnerabilities can also be used to implement `DiscloseByte · WriteByte` interfaces in a similar way by using *heap feng shui* to arrange the overflowed buffer just before a string or array with a length variable that is overwritten with a large value. Next, we show how to leverage our framework once these primitives have been implemented.

6.2. Disarming Same Origin Policy

At this point, the victim has rendered the malicious landing page in a hidden IFRAME from a compromised web site, our embedded JavaScript rendered a number of other domains within the same program address space, and the full functionality of our framework from §4.3.3 is utilized by implementing `DiscloseByte` and `WriteByte` using CVE-2014-1705 or CVE-2015-0359.

These next sections guide one through the necessary navigational steps to obtain all authentication cookies for remote domains, and read, write, and execute DOM contents in a remote domain, as well as make authenticated requests on behalf of the user in a remote domain, which can be used to establish persistence within the browser, among other malicious purposes. For brevity, the next sections detail only the SOP attacks against the sandboxed Chrome browser process within the provided code snippets, but we also briefly mention how the same functionality can be achieved in Firefox and Internet Explorer where the attack varies due to internal browser implementation details. We note as well that nearly all the weaknesses in content isolation logic listed by Zalewski [67][Chapter 9] can be exploited using data-only attacks of the kind presented herein.

6.2.1. Undermining the Security Policy for Cookies. Listing 2 demonstrates one tactic for using our data-only primitives to leak cookie values across domains. In this listing, we have already (i) constructed a memory map offline, (ii) implemented `DiscloseByte` and `WriteByte` using CVE-2014-1705, (iii) followed the relative-offset paths from the memory map *online* to obtain runtime addresses of libraries and heaps for this application instance, and (iv) used process feng shui to render two different domains in the same Chrome sandbox instance. Hence, we know that cookie values for both domains should be present somewhere in the same process address space.

With all these pieces in place, what remains is a simple matter of iterating over the process heap regions to identify the cookies. We do this in two ways. The first method (*scriptCookies*) searches memory using a regular expression that represents the script-accessible cookie structure

3. Because we do not have a more recent vulnerability at hand, we also simulated the CVE-2014-1705 vulnerability in Chrome 50—the latest version as of this writing—and the attacks described herein still work after process feng shui is achieved.

4. Addresses lower than the array address can also be read because overflowing the index results in lower address accesses.

(*e.g., name=value; expires=date;*), while the second method (*requestCookies*) identifies non-script accessible cookies by looking for cookies in request heads (*e.g.,* formatted as *Cookie: name=value*). To ensure that cookies are present in memory as request headers, we can use JavaScript a priori to make requests to the target domain, which automatically adds cookies to requests, including those served over secure connections. Again, as discussed earlier, the code-search facilities (*e.g.,* in Chrome, web searches) made it surprisingly easy to find the variables of interest.

```
// Example 1 - Harvest document.cookie from Heap
var scriptCookies = Heaps(leaked_library).map(
  function(heap) {
    return FindMemory(
      heap.address, heap.length,
      s/(< ... snip cookie regex ... >)/g);
  }
);


// Example 2 - Harvest Cookie Request Header
var requestCookies = Heaps(leaked_library).map(
  function(heap) {
    return FindMemory(
      heap.address, heap.length,
      s/(Cookies: .*)/g);
  }
);
```

Listing 2: Harvesting another domain's cookies, including HTTP-only and secure cookies, using the data-only primitives provided by our framework.

One can envision a number of other tactics using our primitives to disclose not only cookies, but also other sensitive information such as SSL keys, user names, and system configuration. Next, we show one more technique that can not only leak cookies, but also completely circumvent the same origin policy to allow injecting scripts across domains without traditional cross-site scripting (XSS) vulnerabilities, but instead using our data-only primitives.

6.2.2. Undermining Same Origin Policy. The same origin policy is the foundation of web-based security, but Listing 3 demonstrates one way to undermine SOP using data-only primitives to inject scripts across domains. Unlike traditional XSS, which requires a vulnerability in each targeted domain's website, a single memory disclosure vulnerability, combined with our framework, enables one to perform cross-domain script injection to *all* domains.

First, we search the heaps for the *SecurityOrigin* object,[5] which implements the policy-engine for deciding which cross-origin requests are allowed. We search for the values of member variables in the class, which include the port number repeated twice, followed by a boolean value (`m_universalaccess`) which defaults to false. Line 5 of the listing represents the hexadecimal encoding of these values for a domain running on port 8080. Line 6 indicates what to replace the values with, wherein we toggle the `m_universalaccess` variable to *true*. When this variable is true, the domain represented by that *SecurityOrigin* is allowed to access any other domain's DOM.

```
// Step 1 - Modify our SecurityOrigin
Heaps(leaked_ptr).map(function(heap) {
  // SecurityOrigin.m_universalaccess = true
  ReplaceMemory(heap.address, heap.length,
    s/(\x1F\x40\x1F\x40\x00\x00)/g,
      "\x1F\x40\x1F\x40\x00\x01");
});

// Step 2 - Freely manipulate remote DOM
var e =
  newtab.document.createElement("script");
e.text = "alert(document.cookie)";
newtab.document.body.appendChild(e);
```

Listing 3: Setting the `m_universalaccess` member variable of the current web page's *SecurityOrigin* object to *true* and manipulating the remote domain's DOM.

In lines 9–13 we use the universal access capability to append JavaScript code to an arbitrary domain's document-object model, which is subsequently executed in the context of that target domain. Notice that here we are modifying elements in memory *after* they are checked, undermining the security of the DOM policy via a classic time-of-check-to-time-of-use (TOCTTOU) [13] abuse. Additionally, while we only use that cross-domain script injection to leak cookie values in this case, clearly this capability could be used for much more nefarious actions, such as accessing web-based emails, bank accounts, or other services on behalf of the attacker from the victim's own browser. Hence, without true site isolation, "stronger protection for: cookies, documents (html/xml/json), stored passwords, site permissions, and HTML5 stored data" [26] will remain unachievable in practice. We reiterate that this failure is not a Chrome-specific issue, as it is even less hardened in other modern browsers we have examined. We refer the interested reader to the excellent book by Michal Zalewski [67] for a deeper discussion of life outside same origin rules, and the implications thereof.

7. Mitigations

Addressing the underlying problems with the current browser security architecture that permitted our attacks requires support for out-of-process IFRAMEs, but the rollout of that enhancement appears to be a key reason why proper site isolation has taken years to implement. Such code refactoring is not as simple as it may seem at first blush since putting an IFRAME in a different process than its parent page affects a multitude of features in the renderer process, as well as many from the browser process, including navigation, painting, cross-origin scripting, find-in-page, accessibility, printing, to name a few—*"it basically requires a new browser architecture to be implemented, with scores of features updated along the way"* [Charlie Reiss]. In lieu of such architectural changes, we explore other possibilities.

5. See the SecurityOrigin definition.

7.1. Module Compartmentalization

The observant reader would surely have noted that in the proof-of-concept exploits presented earlier, the memory map was used to find a path from a source (the leaked pointer) to a particular destination where some action was carried out (*e.g.,* overwriting security-critical data). Such *cross-module* data accesses, whereby a module reads a data section (either its own or the data section of another module), are significant enabling factors in the attacks described herein. A natural defense, therefore, may be to implement a reference monitor that scrutinizes such cross-module accesses.

To gain deeper insights as to the nature of the observed cross-module accesses, we used Intel's Pin tool to trace the source module of each data access, as well as the target module (*i.e.,* which module's data section was being accessed). Our evaluations show that cross-module data accesses account for over 6% of all benign data accesses we observed in Internet Explorer 11. Similar percentages were observed for Adobe Reader XI. Thus, these results indicate that if these cross-module data access are disallowed, then that might be a solution to curtailing the attacks in this paper. To be fair, at the time of this writing, it is unclear (to us) how much refactoring of complex code would in fact be needed in order to disallow cross-module accesses in the studied applications.

Alternatively, a so-called *micro-service* architecture may offer more well-defined inter-module sandboxing. For instance, each module could run as a separate process with inter-module communications routed through remote procedure calls. This would, of course, prevent an attacker from following module linkages, as address spaces are completely separated. However, this micro-service architecture would require significant code refactoring and there is no obvious method of automating this effort. Thus, we further examined an alternative path that can be used in the near term.

7.2. Global Variable Randomization

A different strategy is to thwart memory cartography altogether by effectively *destroying the map*. In short, an adversary cannot overwrite or disclose target data if navigating to that location from the initial disclosure point is not possible. Recall that the mapping technique takes advantage of the fact that global variables in shared libraries often contain pointers to the code or data of other modules. For example, data is referenced between modules when global data is used in cross-module function parameters, while code may be referenced between modules as event handler parameters. Since global variables are always stored at the same relative offset within a particular library's data section, the adversary can map out the offsets of useful inter-module code and data pointer variables offline, then use those same fixed offsets at runtime to identify the correct global variables used in the memory mapping step.

To thwart an attacker's ability to utilize memory mapping strategies, we explored the design of a new defensive strategy, dubbed global variable layout randomization. In short, the idea is to randomize the order of each global variable in the library (or binary) data section each time that module is loaded. Conceptually, by randomizing variable ordering, the adversary cannot predict the location of variables that contain pointers to other modules, thus thwarting the memory mapping phase.

To provide a solution that could work with commodity binaries, we decided to investigate a binary-level solution. However, randomizing variables at the binary-level presented several non-trivial challenges, including (*i*) determining the location and size of each variable, (*ii*) re-writing code to correctly reference relocated variables, and (*iii*) creating either a new randomized binary or instrumenting operating system loaders to perform randomization in-line as modules are loaded. The first two challenges were addressed by leveraging module metadata which gives information about both *where* code accesses global variables and, through binary code analysis of those locations, *how* those variables are accessed. When source-code is available, this task would be greatly simplified. Additionally, in our proof of concept, we choose to rewrite the PE headers to create a new randomized binary instead of performing the randomization in-line. We leave in-line randomization as an exercise for future work.

7.2.1. Details. Specifically, we first use public debug symbols for each module to obtain a list of variable locations and sizes. Unfortunately, some private variables are not present in these symbols. Hence, our prototype only randomizes those variables readily available in public symbols. Developers could opt-in to sharing the necessary information. One caveat introduced by the unavailability of private symbols, however, is that we need to ensure they are not accidentally randomized, as we have no way of discerning their location or size. To address this issue, we perform an *in-place* randomization, wherein we catalog all available symbols into slots reflecting their size, then randomly swap same-sized variables. In doing so, we guarantee that no undefined (but possibly used) bytes are moved, while still ensuring that known variables are randomized with $n!$ permutations, where n is the number of variables in that slot size. We note that this limitation is alleviated when this same concept is applied at the source level, or in collaboration with developers releasing the appropriate location and size information.

Once global variables are randomized, the next step is to update references to those variables in the binary. Code will access `.data` section variables using an absolute reference (as opposed to a relative reference), and data structures that contain references to global variables use absolute references as well. This is fortunate, as binaries contain a *relocation table* that identifies every absolute reference (in data or code) in the binary. We update each relocation entry that references a randomized global variable to point to the updated variable address.

We prototyped this approach by implementing a custom symbol parser to obtain the necessary symbol information and a binary rewriter to apply the randomizations to PE files on-disk. Firefox and Google Chrome have public symbol information available, and we found that 67%

of *chrome_child.dll* variables and 38% of *xul.dll* variables could be randomized with this swapping approach alone. In both cases, however, the remaining variables are all unique in size and hence could not be swapped in-place. That said, the vast majority of these remaining variables are strings, and hence of limited utility to an attacker attempting to reach additional modules.

Still, when attempting to evaluate the effectiveness of this approach against our attacks, we observed that more than half of the discovered paths in the module where the initial memory disclosure takes place remain unaffected. Consequently, an attacker anticipating this randomization defense may choose to reach only data accessible through unaffected paths. We speculate that this low percentage results from the fact that it is hard to gain reliable information about structs in memory. Our preliminary approach does not assume comprehensive debug symbols, without which we are forced to treat all structs as opaque blocks of data. Consequently, we can only safely swap objects of the same size, leaving objects of distinct sizes untouched.

Given this observation, a possible solution to improve the randomization coverage is *variable relocation*, whereby all identified variables in the `.data` section are moved at random locations within a new section in the executable. By not being able to predict or infer the order of the randomized variables, any leaked pointer into the new section will leave the attacker disoriented. Given that it is possible to safely relocate all variables, this approach will break all cross-module linkages from the module where the initial memory disclosure occurs. We plan to implement and evaluate this improved scheme as part of our future work.

7.3. Other Defenses

Other possibilities might be to better leverage data-flow integrity [20] with complete mediation on memory accesses [34] (for example, to mitigate data-only TOCTTOU abuses like those in §6) to support stronger policies for data. Practical data-plane randomization [8, 23] might also make it more difficult to perform *heap feng-shui*. Undoubtedly, implementations of the aforementioned defenses will certainly increase runtime overhead [54, 61], but in our view, the price seems worth the benefits. Unfortunately, a deeper analysis of tradeoffs in the design and implementation of these techniques is beyond the scope of this paper.

8. Conclusion

In this paper, we introduced a new data-only technique for mapping memory at runtime, and show how that knowledge can be used to undermine the security of modern browsers. Our findings regarding the power of data-only attacks highlight the fact that the lack of proper site isolation in modern browsers undermines many of the efforts made over the past decade to mitigate classes of cross-origin web-based attacks (*e.g.,* cross-site request forgery, reflected XSS, click-jacking, cross-origin resource import) [22, 56]. To address this serious threat, we also proposed first steps towards defensive measures, including global variable randomization. It is our hope that the work in this paper will encourage others to explore additional defenses against data-only attacks before these attacks become far more prevalent.

9. Acknowledgements

We are grateful to the anonymous reviewers for their insightful comments. We also express our gratitude to Niels Provos and Charlie Reis at Google for their detailed comments and suggestions that significantly improved the presentation of this paper. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract no. FA8750-16-C-0199, the Office of Naval Research (ONR) under award no. N00014-15-1-2378, and the National Science Foundation (NSF) awards no. 1421703 and 1617902, with additional support by Qualcomm. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the US government, DARPA, ONR, NSF, or Qualcomm.

References

[1] A Peek into IE's Enhanced Protected Mode Sandbox, Apr 2014.

[2] M. Adabi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[4] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, pages 433–447, 2014.

[5] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security*, pages 1342–1353, 2014.

[6] A. Barth, C. Jackson, C. Reis, and G. C. Team. The security architecture of the Chromium browser. Technical report, Stanford University, 2008.

[7] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, pages 105–120, 2003.

[8] S. Bhatkar and R. Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, 2008.

[9] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *USENIX Security Symposium*, pages 255–270, 2005.

[10] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *ACM Conference on Computer and Communications Security*, pages 268–279, 2015.

[11] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, pages 227–242, 2014.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Asia Conference on Computer and Communications Security*, pages 30–40, 2011.

[13] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *USENIX Security Symposium*, 2005.

[14] E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *IEEE Symposium on Security and Privacy*, pages 243–258, 2014.

[15] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *Symposium on Network and Distributed System Security*, 2016.

[16] M. Brinkmann. Mozilla adds NPAPI plug-in sandbox to Firefox. Sandbox and plug-ins, 2015.

[17] S. Brookes, R. Denz, M. Osterloh, and S. Taylor. Exoshim: Preventing memory disclosure using execute-only kernel code. In *International Conference on Cyber Warfare and Security*, 2016.

[18] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security Symposium*, 2004.

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[20] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[21] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*, pages 559–572, 2010.

[22] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: Get the security of multiple browsers with just one. In *ACM Conference on Computer and Communications Security*, pages 227–237, 2011.

[23] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu. A practical approach for adaptive data structure layout randomization. In *European Symposium on Research in Computer Security*, 2015.

[24] P. Chen, J. Xu, J. Wang, and P. Liu. Instantly obsoleting the address-code associations: A new principle for defending advanced code reuse attack. *preprint arXiv:1507.02786*, 2015.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[26] Chrome Team. Design documents. Sandbox, 2014.

[27] Chrome Team. Site isolation summit: Overview. Overview Videos, 2015.

[28] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy*, pages 763 – 780, 2015.

[29] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *ACM Conference on Computer and Communications Security*, pages 243–255, 2015.

[30] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Symposium on Network and Distributed System Security*, 2015.

[31] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *ACM Asia Conference on Computer and Communications Security*, pages 299–310, 2013.

[32] X. Dong, H. Hu, P. Saxena, and Z. Liang. A quantitative evaluation of privilege separation in web browser designs. In *European Symposium on Research in Computer Security*, pages 75–93, 2013.

[33] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Proceedings of the Internet Measurement Conference (IMC)*, pages 475–488, 2014.

[34] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88, 2006.

[35] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security*, pages 901–913, 2015.

[36] F. Falcon. Exploiting Adobe Flash Player in the era of Control Flow Guard. In *Black Hat Europe*, 2015.

[37] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *Symposium on Network and Distributed System Security*, 2016.

[38] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *ACM Conference on Data and Application Security and Privacy*, pages 325–336, 2015.

[39] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy*, pages 571–585, 2012.

[40] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*, pages 177–192, 2015.

[41] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*, 2016.

[42] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. The "web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing. In *ACM Conference on Computer and Communications Security*, pages 791–804, 2016.

[43] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, 2002.

[44] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2012.

[45] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[46] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *ACM Conference on Computer and Communications Security*, pages 280–291, 2015.

[47] Microsoft Security Team. The enhanced mitigation experience toolkit. EMET, 2016.

[48] D. Moghimi. Subverting without EIP. Overview, 2014.

[49] T. Müller, F. C. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium*, 2011.

[50] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.

[51] J. Obes and J. Schuh. A Tale of Two Pwnies. Part 1, May 2012.

[52] 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*, pages 601–615, 2012.

[53] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, pages 447–462, 2013.

[54] M. Payer. War games ... in memory: shall we play a game?, March 2013.

[55] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, 2003.

[56] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *European Conference on Computer Systems*, 2009.

[57] M. E. Russinovich, D. A. Solomon, A. Ionescu, and M. Pietrek. *Windows internals*. Microsoft, 2009.

[58] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security*, pages 54–65, 2014.

[59] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[60] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

[61] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[62] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security*, pages 256–267, 2015.

[63] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. PixelVault: Using GPUs for securing cryptographic operations. In *ACM Conference on Computer and Communications Security*, pages 1131–1142, 2014.

[64] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principle OS construction of the gazelle web browser. In *USENIX Security Symposium*, 2009.

[65] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM Conference on Computer and Communications Security*, pages 157–168, 2012.

[66] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[67] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[68] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[69] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, pages 337–352, 2013.