RUHR-UNIVERSITÄT BOCHUM
Horst Görtz Institute for IT Security

Technical Report TR-HGI-2016-004

DETILE: Fine-Grained Information Leak Detection in Script Engines

Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, Thorsten Holz

Chair for Systems Security

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

TR-HGI-2016-004
July 29, 2016

RUHR UNIVERSITÄT BOCHUM RUB

hgi Horst Görtz Institut für IT-Sicherheit

DETILE: Fine-Grained Information Leak Detection in Script Engines

Robert Gawlik, Philipp Koppe, Benjamin Kollenda,
Andre Pawlowski, Behrad Garmany Thorsten Holz

Abstract

Memory disclosure attacks play an important role in the exploitation of memory corruption vulnerabilities. By analyzing recent research, we observe that bypasses of defensive solutions that enforce control-flow integrity or attempt to detect return-oriented programming require memory disclosure attacks as a fundamental first step. However, research lags behind when it comes to detecting such information leaks.

In this paper, we tackle this problem and present a system for fine-grained, automated detection of memory disclosure attacks against scripting engines. The basic insight is as follows: scripting languages, such as JavaScript in web browsers, are strictly sandboxed. They must not provide any insights about the memory layout in their contexts. In fact, *any* such information potentially represents an ongoing memory disclosure attack. Hence, to detect information leaks, our system creates a clone of the scripting engine process with a re-randomized memory layout. The clone is instrumented to be synchronized with the original process. Any inconsistency in the script contexts of both processes appears when a memory disclosure was conducted to leak information about the memory layout. Based on this detection approach, we have designed and implemented DETILE (detection of information leaks), a prototype for the JavaScript engine in Microsoft's Internet Explorer 10/11 on Windows 8.0/8.1. An empirical evaluation shows that our tool can successfully detect memory disclosure attacks even against this proprietary software. This impedes *Just-In-Time Code Reuse* and *Counterfeit Object-oriented Programming* attacks.

1 Introduction

Over the last years, many different techniques were developed to prevent attacks that exploit spatial and temporal memory corruption vulnerabilities (see for example the survey by Szekeres et al. [78]). As a result, modern operating systems deploy a wide range of defense methods to impede a successful attack. Examples of these defenses include:

- *Data Execution Prevention* (DEP) [56] is a technique that marks the stack as non-executable and thus an attacker is prohibited from injecting data into a vulnerable application that is later on interpreted as code.
- *Address Space Layout Randomization* (ASLR) [65] means that the memory layout of an application or the kernel is randomized either once during the boot process or every time a process is started. Since the attacker lacks information about the exact memory layout, it is harder for her to predict where her shellcode or reusable code are located.
- *Stack canaries* [22] are random values located on the stack that serve as some kind of guard to hamper memory corruption vulnerabilities.
- Integrity checks of important control structures (e.g., SAFESEH and SEHOP on Windows) protect these control structure from corruptions or enable the detection of ongoing attacks.

Besides these widely deployed techniques, many other defenses were proposed in the literature in the last years [78]. Most notably, the enforcement of *control flow integrity* (CFI) is a promising technique to prevent a whole class of memory corruption vulnerabilities [1]. The basic idea behind CFI is to verify that each control flow transfer leads to a valid target based on a control flow graph that is either statically pre-computed or dynamically generated. Several implementations of CFI with different design constraints, security goals, and performance overheads were published (e.g., [33, 93, 94]).

All these techniques have significantly raised the bar for potential attacks. However, in practice, attackers manage to bypass the deployed defenses and the combination of ASLR and DEP seems to be only a small hurdle [70, 77]. Furthermore, techniques such as *return-oriented programming* (ROP) [48] and its many variants [12, 19, 72] have demonstrated bypasses of many existing and proposed defenses. Recently, several papers demonstrated bypasses of existing CFI solutions [29, 38]. As such, there is a mismatch between the (theoretical) security guarantees provided by a given defense and its practical implementation: while the combination of ASLR and DEP implies that an attacker has little knowledge about the memory layout, she can find ways to bypass the deployed defenses with an iterative approach.

A general observation is that the first step in modern attacks is based on a *memory disclosure attack* (also referred to as *information leak*): the adversary finds a way to read a (raw) memory pointer to learn some information about the virtual address space of the vulnerable program. Generally speaking, the attacker can then de-randomize the address space based on this leaked pointer (thus bypassing ASLR), use ROP to bypass DEP, and finally execute shellcode of her choice. Modern exploits leverage information leaks as a fundamental primitive. Furthermore, recent CFI and ROP defense bypasses use memory disclosures as well. For example, Snow et al. introduced *Just-In-Time Code Reuse* attacks (JIT-ROP [73]) to bypass fine-grained ASLR implementations by repeatedly utilizing an information leak. Similarly, Snow et al. were able to circumvent approaches which incorporate *destructive code reads*, a mechanism to prevent execution of code after it has been read [74]. *G-Free* [59], a compiler-based approach against any ROP attack, was recently circumvented by Athanasakis et al. [4]. Their technique requires successive information leaks to disclose enough needed information. Göktaş et al. demonstrated several bypasses of proposed ROP defenses and their exploit needs an information leak as a first step [39]. An information leak is also needed by Song et al., who showed that dynamic code generation is vulnerable to code injection attacks [75]. Similarily, *Counterfeit Object-oriented Programming* (COOP [66]) needs to disclose the location of *vtables* to mount a subsequent control-flow hijacking attack by reusing them. Disclosures are also utilized by *memory oracles* to weaken various defenses [37]. *All* of these offensive bypasses utilized an information leak as a first step and implemented the attack against a web browser.

Another general observation is that script engines in web browsers are commonly utilized by adversaries to abuse information leaks in practice. Web browsers have evolved into complex systems that even build the base for a new era of operating systems, sitting on top of the kernel (e.g., FirefoxOS or ChromeOS). This implies that browsers are an attractive target for attacks. Browser vulnerabilities are prevalent and as the yearly *pwn2own* competition shows, researchers successfully use them to take control of the machine. Notably, most of these attacks are based on vulnerabilities that create an information leak utilizing the script engine. In summary, we see a lot of achievements on the offensive side (especially targeting browsers) but research lags behind when it comes to detecting such information leaks.

In this paper, we take these observations into account and propose a technique for fine-grained, automated detection of memory disclosure attacks against script engines at runtime. Our approach is based on the insight that information leaks are leveraged by state-of-the-art exploits to learn the placement of modules—and thereby code sections—in the virtual address space in order to bypass ASLR. Any sandboxed script context is forbidden to contain memory information, i.e., no script variable is allowed to provide a memory pointer. As such, a viable approach to detect information leaks is to create a clone of the to be protected process with a re-randomized address space layout, which is instrumented to be synchronized with the original process. An inconsistency in the script contexts of both processes can only occur when a memory disclosure vulnerability was exploited

Protection flavor	Defense	Weakened/Bypassed by
Address randomization	Fine-grained ASLR [43]	Just-In-Time Code Reuse [73]
Code-reuse protection	RopGuard [35] KBouncer [62] ROPecker [20]	Size Does Matter [39], Anti-ROP Evaluation [67], COOP [66]
Code-reuse protection	G-Free [59]	Browser JIT Defense Bypass [4], COOP [66]
Coarse-grained CFI	CCFIR [93] BinCFI [94]	Stitching the Gadgets [29], Out of Control [38], COOP [66]
Fine-grained CFI	IFCC [80] VTV [80]	Losing Control [21]
Information hiding	Oxymoron [5]	Vtable Disclosure [30], Crash Resistance [37], COOP [66]
Information hiding	CPI linear region [49]	Crash Resistance [37]
Execution randomization	Isomeron [30]	Crash Resistance [37]
Randomization/ Information hiding	Readactor [24]	Crash Resistance [37], COOP [66]
Randomization/ Destructive code reads	Heisenbyte [79] NEAR [85]	Code Inference Attacks [74]

Table 1: Proposed defenses and offensive approaches utilizing an information leak in browsers to weaken or bypass the specific defense. All mentioned attacks are mitigated by DETILE.

to gain information about the memory layout. In such a case, the two processes can be halted to prevent further execution of the malicious script. An overview of bypassed defenses by specific attacks which are mitigated by our approach is shown in Table 1. In spirit, our approach is similar to n-variant systems [14, 23] and similar multi-execution based approaches [16, 26, 31]. However, we are able to observe the actual information leak since we instrument the scripting context, while n-variant systems are only capable of observing when the control flow diverges in the different replica. As such, we can detect modern code-reuse attacks such as JIT-ROP [73] or COOP [66].

We have implemented a prototype of our technique in a tool called DETILE (detection of information leaks). We extended Internet Explorer 10/11 (IE) on Windows 8.0/8.1 to create a synchronized clone of each tab and enforce the information leak checks. We chose this software mainly due to two reasons. First, IE is an attractive target for attackers as the large number of vulnerabilities indicates. Second, IE and Windows pose several interesting technical challenges since it is a proprietary binary system that we need to instrument and it lacks fine-grained ASLR. Evaluation results show that our prototype is able to re-randomize single processes without significant computational impact. Additionally, running IE with our re-randomization and information leak detection engine imposes a performance hit of ∼17% on average. Furthermore, empirical tests with real-world exploits also indicate that our approach is usable to unravel modern and unknown exploits which target browsers and utilize memory disclosures.

In summary, our main contributions in this paper are:

- We present a system to tackle the problem of information leaks, which are frequently used in practice by attackers as an exploit primitive. More specifically, we propose a concept for fine-grained, automated detection of information leaks with per process re-randomization, dual process execution, and process synchronization.
- We show that dual execution of highly complex, binary-only software such as Microsoft's Internet Explorer is possible without access to the source code, whereby two executing

instances operate deterministic to each other.

- We implemented a prototype for IE 10/11 on Windows 8.0/8.1. We show that our tool can successfully detect several real-world exploits, while producing no alerts on highly complex, real-world websites.

2 Technical Background

Before diving into the details of our defense, we first review several technical details of Windows related to the implementation of ASLR, the interplay between 64- and 32-bit processes, and the architecture of IE. This information is based on empirical tests we performed and reverse engineering of certain parts of Windows and IE. We briefly introduce important scripting engines and also explain the attacker model used throughout the rest of this paper.

2.1 Enhancing Security with N-Variant Systems

N-Variant or *Multi-Execution* systems evolved from fault-tolerant environments to mitigation systems against security critical vulnerabilities [14,23,44,81]. Our concept of DETILE incorporates similar ideas like dual process execution and dual process synchronization. However, our approach is constructed specifically for scripting engines, and thus, is more fine-grained: While DETILE operates and synchronizes processes on the scripting interpreter's bytecode level, n-variant systems intercept only at the system call level. One drawback for these conventional systems is that they are prone to *Just-In-Time Code-Reuse* (JIT-ROP [73]) and *Counterfeit Object-oriented Programming* (COOP [66]) attacks, while DETILE is able to detect these (see Sections 3.1 and 6.5 and for details).

2.2 Windows ASLR Internals

Address Space Layout Randomization (ASLR) is a well-known security mechanism that involves the randomization of stacks, heaps, and loaded images in the virtual address space. Its purpose is to leave an attacker with no knowledge about the virtual memory space in which code and data lives. Combined with DEP, ASLR makes remote system exploitation through memory corruption techniques a much harder task. While brute-force attacks against services that automatically restart are possible [9], such attacks are typically not viable in practice against web browsers.

In Windows, whenever an image is loaded into the virtual address space, a section object is created, which represents a section of memory. These objects are managed system-wide and can be shared among all processes. Once a DLL is loaded, its section object remains permanent as long as processes are referencing it. This concept has the benefit that relocation takes place once and whenever a process needs to load a DLL, its section object is reused and the view of the section is mapped into the virtual address space of the process, making the memory section visible. This way, physical memory is shared among all processes that load a specific DLL whose section object is already present. In particular, as long as the virtual address is not occupied, each image is loaded at the same virtual address among all running usermode processes. Figure 1 illustrates this concept. The randomization of a DLL is influenced by a random value (the so called *image bias*) that is generated at boot time. This value is used as an index in an image bitmap, which represents specific address ranges. For 32-bit images, the top address of the range starts at 0x78000000. For 64-bit images that are based above the 4GB boundary, the top address of the range starts at 0x7FFFFFE0000. Each bit in the bitmap stands for a 64KB unit of allocation starting from the top address to lower addresses. When an image is being loaded, the bitmap is scanned from top to bottom starting at the random image bias until enough free bits are found to map the image. In Windows 8, there are three image bitmaps. One is for 64-bit images above the 4GB address range, one for 64-bit images below 4GB, and the third bitmap is used for 32-bit images.

64-bit DLL images that are based above the 4GB address boundary receive 19 bits of entropy. It is worth mentioning that prior to Windows 8, the ASLR entropy amounted to 8-bit and was the

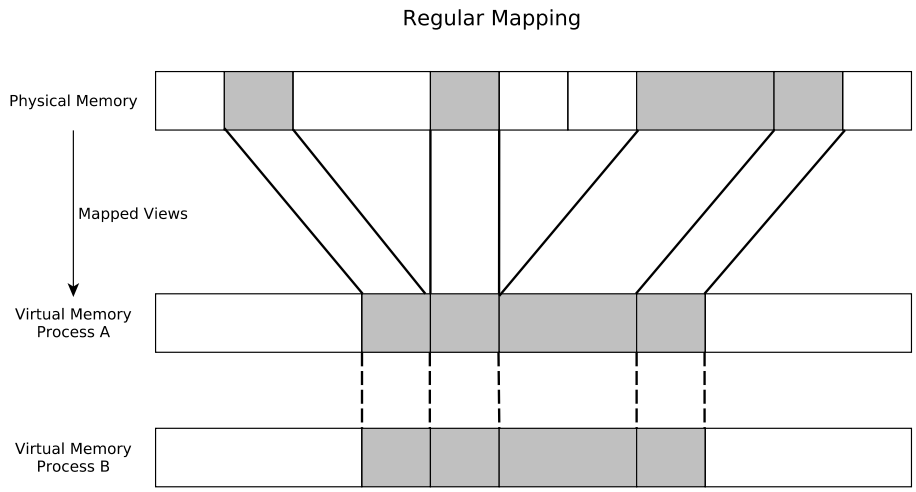

Figure 1: Shared physical memory: shaded regions are sections of memory occupied for images. Their views are mapped into the virtual address space of the processes that load the images.

same for both 32-bit and 64-bit images. Executable images other than DLLs receive an entropy of 17 bits when they are based above 4GB, otherwise they receive 8 bits.

Whenever an executable image is loaded, a random load offset is calculated which corresponds to the entropy the image receives. Thus, an executable image might get relocated to another base once the last reference to its image section is gone. However, Windows does not discard the image section object immediately, but rather keeps it in case the image is loaded soon after. This leads to the empirical fact that executable images are loaded at the same base as before.

While all these features make sense from a performance point of view, they create an inconvenient state for our implementation and detection metric. As we discuss in Section 4, we rebase each DLL and the main executable for each run.

2.3 WOW64 Subsystem Overview

64-bit operating systems are the systems of choice for today's users: 64-bit processors are widely used in practice, and hence Microsoft Windows 7 and later versions are usually running in the 64-bit version on typical desktop systems. However, most third-party applications are distributed in their 32-bit form. This is for example the case for Mozilla Firefox, and also for parts of Microsoft's Internet Explorer. As our framework should protect against widely attacked targets, it needs to support 32-bit and 64-bit processes. Therefore, the *Windows On Windows 64* (shortened as *WOW64*) emulation layer plays an important role, as it allows legacy 32-bit applications to run on modern 64-bit Windows systems.

Executing a user-mode 32-bit application instructs the kernel to create a WOW64 process. According to our observations, it creates the program's address space and maps the 64-bit and 32-bit *NT Layer DLL* (`ntdll.dll`) into the virtual memory of the program. The 64-bit `ntdll.dll` is mapped to an address greater than 4GB, and the 32-bit `ntdll.dll` to an address smaller than 4GB. Then, the application's 32-bit main executable is mapped into memory. These three images are the modules, which are available in a user-mode address space, even when starting a 32-bit application in suspended mode. Resuming the application leads to the mapping of the emulation layer dynamic link libraries `wow64.dll`, `wow64cpu.dll` and `wow64win.dll`. They manage the creation of 32-bit processes and threads, enable CPU mode switches between 32-bit and 64-bit during system calls, and intercept and redirect 32-bit system calls to their 64-bit equivalents. For more details about the WOW64 layer, the reader is referred to literature on Windows Internals [65]. Subsequent 32-bit DLLs are mapped into the address space via `LdrLoadDll` of the 32-bit `ntdll.dll`. The first of them is `kernel32.dll`. The loader assures that it is mapped to the same address in each

WOW64 process system wide, using a unique address per reboot. It therefore compares its name to the hardcoded "KERNEL32.DLL" string in `ntdll.dll` upon loading. If the loader is not able to map it to its preferred base address, process initialization fails with a conflicting address error. As process based re-randomization plays a crucial role in our framework, this issue is handled such that each process contains its `kernel32.dll` at a different base address (see Section 4.1). After mapping `kernel32.dll`, all other needed 32-bit DLLs are mapped into the address space by the loader via the library loading API. System libraries are thereby normally taken from the `C:\Windows\SysWOW64` folder that comprises the counterpart of `C:\Windows\System32` for 32-bit applications.

2.4 Internet Explorer Architecture

While our approach is in general applicable to other software, we focus on protecting the scripting engines of a recent version of Microsoft Internet Explorer since browsers are one of the most common targets. Additionally, IE is a high value target as demonstrated by the number of code execution vulnerabilities compared to other browsers [27,28]. As we will frequently refer to browser internals, a basic understanding of its architecture is needed.

Since version 8, IE is developed as multi-process application [92]. That means, a 64-bit main frame process governs several 32-bit WOW64 tab processes, which are isolated from each other. The frame process runs with a medium integrity level and isolated tab processes run with low integrity levels. Hence, tab processes are restricted and forbidden to access all resources of processes with higher integrity levels [55]. This architecture implies that websites opened in new tabs can lead to the start of new tab processes. These have to incorporate our protection in order to protect IE as complete application against information leaks (see Section 4).

2.5 Scripting Engines

In the context of IE, mainly two scripting engines are relevant and we briefly introduce both.

2.5.1 Internet Explorer Chakra

With the release of Internet Explorer 9, a new JavaScript engine called *Chakra* was introduced. Since Internet Explorer 11, Chakra exports a documented API which enables developers to embed the engine into their own applications. However, IE still uses the undocumented internal COM interface. Nevertheless, some Chakra internals were learned from the official API. The engine supports just-in-time (JIT) compiling of JavaScript bytecode to speed up execution. Typed arrays like integer arrays are stored as native arrays in heap memory along with metadata to accelerate element access. Script code is translated to JS bytecode on demand in a function-wise manner to minimize memory footprint and avoid generating unused bytecode. The bytecode is interpreted within a loop, whereby undocumented *opcodes* govern the execution of native functions within a switch statement. Dependent on the opcode, the desired JavaScript functionality is achieved with native code.

2.5.2 ActionScript Virtual Machine (AVM)

The *Adobe Flash* plugin for browsers and especially for IE is a widely attacked target. Scripts written in *ActionScript* are interpreted or JIT-compiled to native code by the AVM. There is much unofficial documentation about its internals [10, 52]. Most importantly, it is possible to intercept *each* ActionScript method with available tools [42]. Thus, no matter whether bytecode is interpreted by the opcode handlers or JIT code is executed, we are able to instrument the AVM.

2.6 Adversarial Capabilities

Memory disclosure attacks are an increasingly used technique for the exploitation of software vulnerabilities [71, 73, 77]. In the presence of full ASLR, DEP, CFI, or ROP defenses, the attacker

has no anchor to a memory address to jump to, even if in control of the instruction pointer. This is the moment where information leaks come into play: an attacker needs to read—in any way possible—a raw memory pointer in order to gain a foothold into the native virtual address space of the vulnerable program.

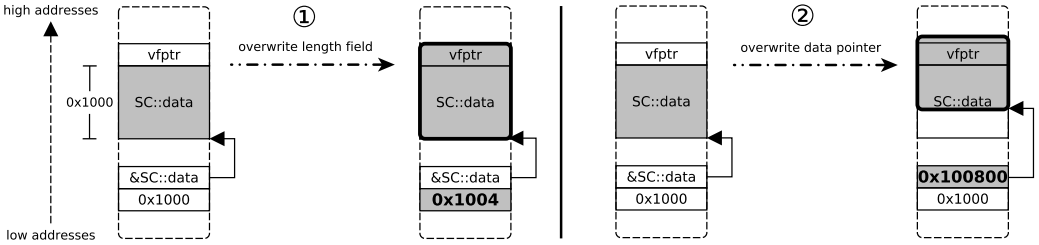

Figure 2: Two sketched methods to achieve information leaks: 1.) Overwriting a length field (`0x1000`) of a script context data structure gives the adversary the possibility to read beyond legitimate data (`SC::data`) and leak the address of a vtable (`vfptr`). 2.) When the pointer (`&SC::data`) to the data structure is modified directly, it can point into the data structure and can disclose memory beyond the legitimate data.

One common way to achieve an information disclosure of native memory is to use a vulnerability to eventually overwrite a data length field, without crashing the program. The next step is an out-of-bounds read on the underlying data, to subsequently read memory information. The field and information may have been provoked to reside in predictable locations by heap massaging in the script context, performed by the attacker. Another possibility to disclose memory is to use a program's vulnerability to write a memory pointer into data that must not provide memory information, such as a string in the script context. Similarly, overwriting a terminating character of a script context data structure (e.g., a wide char null of a JavaScript string) leads to a memory disclosure, as subsequent memory content (i.e., after the string data) is presented to the attacker when reading this data structure.

New powerful scripting features also found their way into the development arsenal of attackers [87, 88]: typed arrays [40] make it possible to read and write data very fine-grained within a legitimate scripting context. Manipulating either a length field or a pointer to an array buffer directly inside the metadata can lead to full read and write access of the process' memory. Figure 2 sketches only two general schemes of many possibilities to create information leaks. Note that leaks can also occur due to uninitialized variables or other errors and do not have to be created like shown in Figure 2. As soon as the attacker can read process memory, she can learn the base addresses of loaded modules in the address space of the program. Then, any code-reuse primitives can be conducted to exploit a vulnerability in order to bypass DEP, ASLR, CFI [29] and ROP defenses [17, 39]. Another possibility is to leak code directly to initiate an attack and bypass ASLR [73]. Other mitigations like Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [54] cannot withstand capabilities of sophisticated attackers.

For applications with scripting capabilities, untrusted contexts are sandboxed (e.g., JavaScript in web browsers) and must not provide memory information. Thus, attackers use different vulnerabilities to leak memory information into that context [38, 71, 86]. We assume that the program we want to protect suffers from such a memory corruption vulnerability that allows the adversary to corrupt memory objects. In fact, a study shows that *any* type of memory error can be transformed into an information leak [78]. Furthermore, we assume that the attacker uses a scripting environment to leverage the obtained memory disclosure information at runtime for her malicious computations. This is consistent with modern exploits in academic research [17, 29, 38, 39, 67] as well as in-the-wild [68, 82, 86–88]. Our goal is to protect script engines against such powerful, yet realistic adversaries.

Thus, information leaks are an inevitable threat even in the presence of state-of-the-art secu-

rity features. Note that many use-after-free vulnerabilities can be transformed into information leaks [87]. Thus, especially web browser are in high danger as these errors are prevalent in such complex software systems.

3 System Overview

In the following, we explain our approach to tackle the challenge of detecting information leaks in script engines. Hence, we introduce the needed building blocks, namely per process re-randomization and dual process execution.

3.1 Main Concept

As described above, information leaks manifest themselves in the form of memory information inside a context which must not reveal such insights. In our case, this is any script context inside an application: high-level variables and content in a script must not contain memory pointers, which attackers could use to deduce image base addresses of loaded modules.

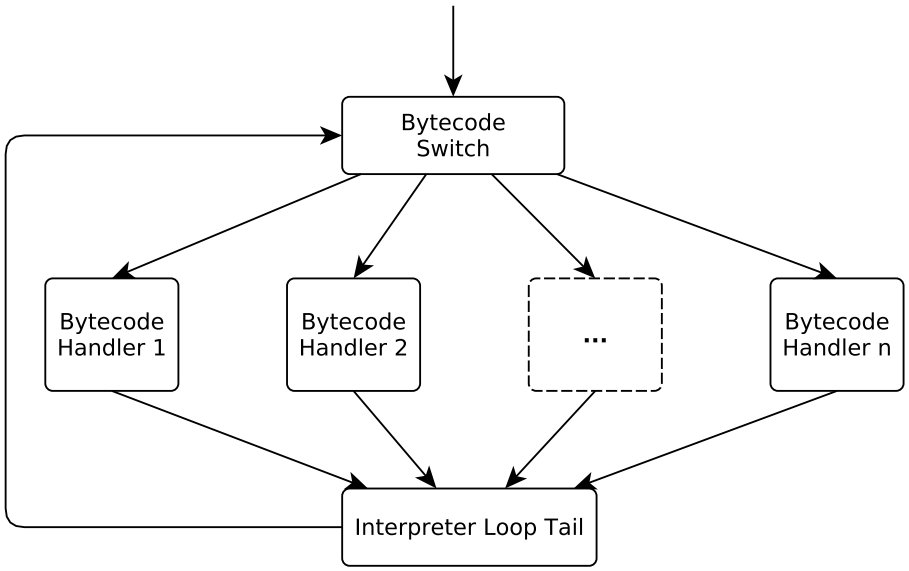

Figure 3: The basic concept of a script engine's interpreter loop. The interpreter fetches the bytecode of the script and switches to the corresponding bytecode handler. When the operation of the bytecode handler is finished, the interpreter loop jumps back to the switch and processes the next bytecode.

Unfortunately, a legitimate number and a memory pointer in data bytes received via a scripting function are indistinguishable. This leads us to the following assumption: a memory disclosure attack yields a memory pointer, which may be surrounded by legitimate data. The same targeted memory disclosure, when applied to a differently *randomized*, but otherwise *identical* process, will yield the same legitimate data, but a *different* memory pointer. Due to the varying base addresses of modules, different heap and stack addresses, a memory pointer will have a different address in the second process than in the first process. Thus, a master process and a cloned twin process—with different address space layout randomization—can be executed synchronized side-by-side and perform identical operations, e.g., execute a specific JavaScript function. In benign cases, the same data getting into the script context is equal for both processes. When comparing the received data of one process to the same data received in the second process, the only difference can arise because of a leaked memory pointer pointing to equal memory, but having a *different* address. In order to compare the data of the master and twin process, we have to instrument the interpreter loop of the script engine. Figure 3 shows the basic concept of a script engine's interpreter loop. We can

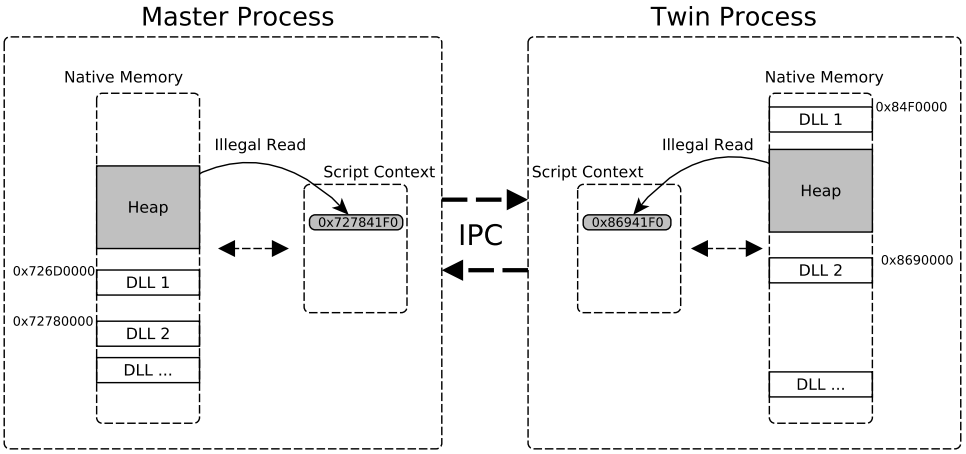

Figure 4: Overview of our main information leak detection concept: The master process is synchronized with a re-randomized, but otherwise identical twin process. If a memory disclosure attack is conducted in the master, it appears as well in the twin. Due to the different randomization, the disclosure attack manifests itself in different data flowing into the script context and can be detected (`0x727841F0` vs. `0x86941F0`)

instrument the `call` and `return` bytecodes to precisely check all outgoing data and therefore to detect an information leak.

Based on this principle, our prototype system launches the same script engine process twice with diverse memory layouts (see also Figure 4). The script engines are coupled to run in sync which enables checking for information leaks. In spirit, this is similar to n-variant systems [14, 23] and multi-execution based approaches [16, 26, 31]. However, our approach is more fine-grained since it checks and synchronizes the processed data on the bytecode level of the script context and is capable of detecting the actual information leak, instead of merely detecting an artifact of a successful compromise (i.e., divergence in the control flow). A more detailed discussion about the granularity of our approach in comparison to other n-variant and multi-execution systems is given in Section 6.5. The involved technical challenges to precisely detect information leaks are explained throughout the rest of this paper.

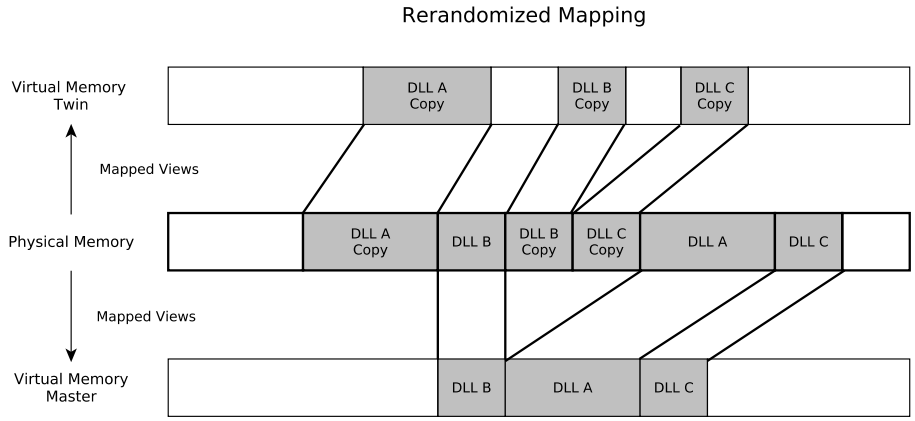

Figure 5: Master and twin process with a different randomization: as the loader has to fix up address references for each twin process, sharable code turns into private data. As a consequence, each twin process has its own private copy of the DLL.

3.2 Per Process Re-randomization

As sketched in Section 2.2, all executable images loaded among simultaneously running processes have the same base address in these processes. While it is convenient from a memory sharing point of view, an attacker can abuse a memory disclosure for coordinated attacks between them [51]. Applying a different randomization for processes of choice has the nice side effect of excluding these from such attacks, but our per process re-randomization has the main goal to randomize two running instances of the *same* program (see Figure 5). Therefore, a program of interest is started and we collect the base addresses of all images it loads and will load during its runtime period. We refer to this first process as *master* process. A second process instance of the application known as the *twin* process is spawned. Upon its initialization, the base addresses gained from the master are occupied in the virtual address space of the twin. This forces the image loader to map the images to other addresses than in the master process, as they are already allocated. We can save us time and trouble to re-randomize the stack and heap process-wise, as modern operating systems (e.g., Windows 8 on 64-bit) support it natively. Though, the steps described to re-randomize all loaded images in the twin process are specific to Windows, the general concept of our proposed information leak detection approach is operating system independent. As long as master and twin process have different memory layouts, our approach can be applied. Finally, we can establish an *inter-process communication* (IPC) bridge between the master and twin process. This enables synchronized execution between them and comparison of data flows into their contexts that are forbidden to contain memory information.

3.3 Dual Process Execution and Synchronization

After the re-randomization phase, both processes are ready to start execution at their identical entrypoints. After exchanging a handshake, both resume execution. In order to achieve comparable data for information leak checking, the executions of script interpreters in both processes have to be synchronized precisely. This is accomplished by intercepting an interpreter's native methods. Additionally, we install hooks inside the bytecode interpreter loop at positions where opcodes are interpreted and corresponding native functions are called. Thus, we perceive any high-level script method call at its binary level. The master drives execution and these hooks are the points where the master and twin process are synchronized via IPC. We check for information leaks by comparing binary data which returns as high-level data into the script context. All input data the master loads are stored in a cache and replayed to the twin process to ensure they operate on the same source (e.g., web pages a browser loads). Built-in script functions that potentially introduce entropy (e.g., `Math.random`, `Date.now`, and `window.screenX` in JavaScript) interfere with our deployed detection mechanism, since they generate values inside the script context that are different from each other in the master and twin processes, respectively. Additionally, they may induce a divergent script control flow. Both occurrences would be falsely detected as memory disclosure. Thus we also synchronize the entropy of both processes by copying the generated value from the master to the twin process. This way the twin process continues working on the same data as the master process and we are creating a co-deterministic script execution.

4 Implementation Details

Based on the concepts of per process re-randomization and dual process execution, we implemented a tool called DETILE for Windows 8.0 and 8.1 64-bit. The current prototype is able to re-randomize on a per process basis and instrument Internet Explorer 10 and 11 to run in dual process execution mode. In the following, we describe in detail the steps taken during the development of our framework to detect information leaks and also discuss arosen and solved challenges.

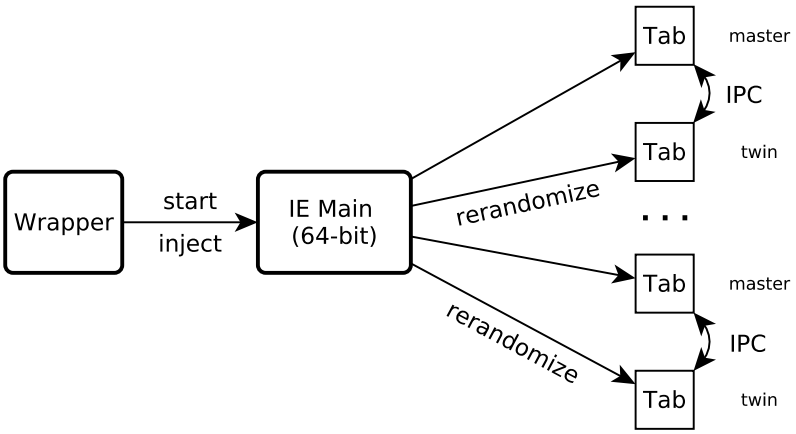

Figure 6: DETILE running with Internet Explorer. A 64-bit duplicator library is injected into the main IE frame process to enable it creating and rerandomizing twin tab processes for each master tab process, by itself. The main IE frame also injects a 32-bit DLL into each tab process to allow synchronization, communication between master and twin, and information leak detection.

4.1 Duplication and Re-randomization

In order to re-randomize processes and load images at different base addresses, we developed a duplicator which creates a program's master process. It enumerates the master's initial loaded images with the help of the Windows API (`CreateToolHelp32Snapshot`) before the master starts execution. Then, the twin process is created in suspended mode, and a page is allocated in the twin at all addresses of previously gathered image bases. We then need to trick the Windows loader into mapping `kernel32.dll` at a different base in the twin. Therefore, the twin is attached to the DebugAPI and a breakpoint is set automatically to the function `RtlEqualUnicodeString` in the 32-bit loader in the NT Layer DLL. The twin is then resumed and the WOW64 subsystem DLLs are initialized successfully to different base addresses, at first. As soon as the breakpoint triggers, and the function tries to compare the unicode name "KERNEL32.DLL" to the hardcoded "KERNEL32.DLL" string in NT Layer DLL, the arguments to `RtlEqualUnicodeString` are modified: the first unicode name is changed to lowercase and the third parameter is set to perfom a case sensitive comparison. This way, the loader believes that a *different* DLL than `kernel32.dll` is going to be initialized and allows the mapping to a different base. The loading of the 32-bit `kernel32.dll` is performed immediately after the WOW64 subsystem is initialized and it is also the first 32-bit DLL being mapped after the 32-bit NT Layer DLL. Thus, all subsequent libraries that are loaded and import functions from `kernel32.dll` have no problems to resolve their dependencies using the remapped `kernel32.dll`. The loader maps them to different addresses, as their prefered base addresses are reserved. Although the DebugAPI is used, all steps run in a fully automated way. As a next step, the DebugAPI is detached and the main image is remapped to a different address. As it is already mapped even in suspended processes, this has to be done specifically. Additionally, `LdrLoadDll` in the twin process is detoured to intercept new library loads and map incoming images to different addresses than in the master.

We were not able to re-randomize `ntdll.dll` because it is mapped into the virtual address space very early in the process creation procedure. Attempts to remap `ntdll.dll` later on did not succeed due to callbacks invoked by the kernel. The implications of a non re-randomized `ntdll.dll` are discussed in Section 7.2.

Note that this design works also with pure 64-bit processes. However, frequently attacked applications like tab processes of Internet Explorer are 32-bit and are running in the WOW64 subsystem. Hence, our framework has to protect them as well. The following explains how DETILE achieves this support.

While the above explained logic is sufficient to duplicate and re-randomize a single-process

program, additional measures have to be taken in the case of multi-process architecture applications like Internet Explorer. Therefore, we developed a wrapper which starts the 64-bit main IE frame process and injects a 64-bit library, which we named duplicator library (see Figure 6). This way, we modify the frame process, such that each time a tab process is started by the frame process, a second tab process is spawned. The first becomes the master, the second the twin. This is achieved via detouring and modifying the process creation of the IE frame. Additionally, our above explained re-randomization logic is incorporated into the duplicator library to allow the main IE frame process itself to re-randomize its spawned twins at creation time. To protect each new tab which is run by the IE frame, we ensure that each tab is run in a new process and gets a twin. To enable communication, synchronization, and detection of information leaks, the duplicator injects also a 32-bit library into the master and the twin upon their creation by the main IE frame process.

4.1.1 Kernel Mode Approach

In addition to our user-mode approach, we also developed a kernel driver that follows the same logic. The driver rebases all DLLs and the main image, except for `ntdll.dll`. The main benefit of approaching the problem from kernel mode is flexibility. It enables us to intercept and filter each process and image load and grants us access to internal data structures that are linked to each image. The driver also handles images that are dynamically loaded, no matter through which API call the request is triggered. This is important as we noticed that not all DLL mappings go through the native `LdrLoadDll` call. Another motivation for a kernel approach is its generic functionality, in that we are not bound to apply a logic tailored to a specific process, but to apply one logic for each process. However, we left the generic functionality as a future work.

4.2 Synchronization

We designed our prototype to be contained in a DLL which is loaded into both target instances. To reliably intercept all script execution, we hook `LdrLoadDll` to initialize our synchronization as early as possible once the engine has been loaded. After determining the role (master or twin), the processes exchange a short handshake and wait for events from the interpreter instrumentation. While most of our work is focused on the scripting engine, we also instrument parts of `wininet.dll` to provide basic proxy functionality. The twin receives an exact copy of the web data sent to the master to ensure the same code is executed.

4.2.1 Entropy Normalization

The synchronization of script execution relies heavily on the identification of functions and objects introducing entropy into the script context. Values classified as entropy are overwritten in the twin with the value received from the master. This ensures that functions such as `Math.random` and `Date.now` return the exact same value, which is crucial for synchronous execution. While it is obvious for `Date.now`, it is not immediately clear for other methods. Therefore, *entropy inducing* methods are detected and filtered incremetally during runtime. Hence, if a detection has triggered but the cause was not an information leak, it is included into the list of entropy methods.

4.2.2 Rendezvous and Checking Points

Vital program points where master and twin are synchronized are bytecode handler functions. If a handler function returns data into the script context, it is first determined if the handler function is an entropy inducing function. However, the vast majority of function invocations and object accesses do not introduce entropy and are checked for equality between master and twin on the fly. If a difference is encountered that is not classified as entropy, we assume that an information leak occurred and take actions, namely logging the incident and terminating both processes. Our empirical evaluation demonstrates that the synchronization is precise and even for complex websites, we can synchronize the master and twin process (see Section 5.2 for details).

4.3 Chakra Instrumentation

The Chakra JavaScript Engine contains a JIT compiler. It runs in a dedicated thread, identifies frequently executed (so called *hot*) functions and compiles them to native code. Our current implementation works on script interpreters, hence we disabled the JIT compiler. This is currently a prototype limitation whose solution we discuss in Section 7.2.

In order to synchronize execution and check for information leaks, we instrumented the main loop of the Chakra interpreter, which is located in the `Js::InterpreterStackFrame::Process` function. It is invoked recursively for each JavaScript call and iterates over the variable length bytecodes of the JavaScript function. The main loop contains a `switch` statement, which selects the corresponding handler for the currently interpreted bytecode. The handler then operates on the JavaScript context dependent on the operands and the current state. In the examined Chakra versions, we observed up to 648 unique bytecodes. Prior to the invocation of a bytecode handler, our instrumentation transfers the control flow to a small, highly optimized assembly stub, which decides whether the current bytecode is vital for our framework to handle.

We intercept all `call` and `return` as well as necessary `conversion` bytecodes in order to extract metadata such as JavaScript function arguments, return values, and conversion values. `Conversion` bytecodes handle dynamic type casting, native value to JavaScript object and JavaScript object to native value conversions. Additionally, we intercept engine functions that handle implicit type casts at native level, because they are invoked by other bytecode handlers as required and have no bytecode equivalents themselves. Furthermore, all interception sites support the manipulation of the outgoing native value or JavaScript object for the purpose of entropy elimination in the JavaScript context of the twin process.

4.4 AVM Instrumentation

Instrumentation of the AVM is based on prior work of F-Secure [42] and Microsoft [52]. We hook at the end of the native method `verifyOnCall` inside `verifyEnterGPR` to intercept ActionScript method calls and retrieve ActionScript method names. At these points, master and twin can be synchronized. Parameters flowing into an ActionScript method and return data flowing back into the ActionScript context can be dissected, too. They are also processed inside the method `verifyEnterGPR`. Based on their high level ActionScript types, the parameters and return data can be compared in the master and twin. This way, we can keep the master and twin in sync at method calls, check for information leaks and mediate entropy data from the master to the twin.

5 Evaluation

In the following, we present evaluation results for our prototype implementation of Detile in the form of performance and memory usage benchmarks. The benchmarks were conducted on a system running Windows 8.0/8.1 that was equipped with a 4th generation Intel i7-4710MQ quad-core CPU and 8GB DDR3 RAM. Furthermore, we demonstrate how our prototype can successfully detect several kinds of real-world information leaks.

5.1 Re-randomization of Process Modules

We evaluated our re-randomization engine according to its effectiveness, memory usage, and performance.

5.1.1 Effectiveness

We applied re-randomization to internal Windows applications and third-party applications, to verify that modules in the twin are based at different addresses than in the master. We therefore compared base addresses of all loaded images between the two processes and confirmed that all images in the twin process had a different base address than in the master, except `ntdll.dll`. See

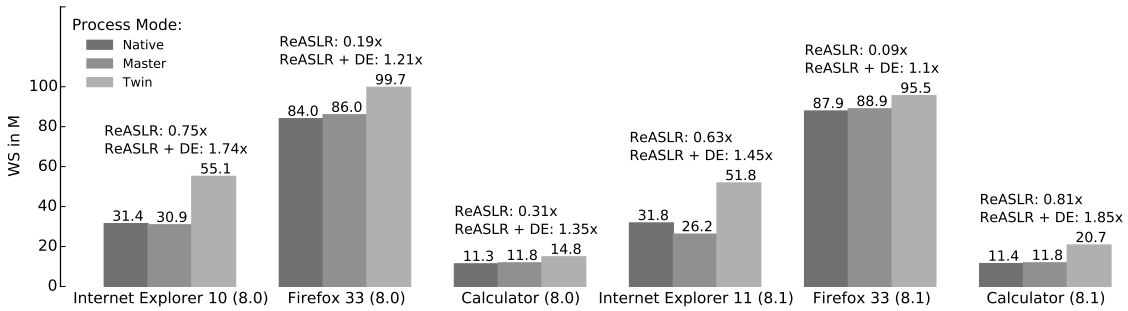

Figure 7: Memory overhead of re-randomization and dual execution measured via working set (WS) consumption in megabytes (M): Native processes on Windows 8.0 and 8.1 are contrasted to their counterparts running in re-randomized dual execution mode (master and twin).

the discussion in Section 7.2 for details on the difficulties of remapping the 64-bit and 32-bit NT Layer DLLs. Table 4 in Appendix 9.1 lists important Windows DLLs, re-randomized in different processes running *simultaneously* on a *single* user session.

5.1.2 Physical Memory Usage

To inspect the memory overhead of our re-randomization scheme, we measured the working set characteristics for different master and re-randomized twin processes compared to native processes. Figure 7 shows the memory working sets of three applications. *ReASLR* denotes thereby the re-randomization within a single process. *DE* means that two processes are running, whereby the master's randomization is kept native while the twin is re-randomized. The applications besides IE are only included to measure the memory overhead and are not synchronized. We calculate the memory overhead of per process re-randomization (*ReASLR*) of a *single* process as follows:

$$Overhead(ReASLR) = \frac{WS(Twin)}{WS(Native)} - 1$$

Thus, the overall memory overhead based on working sets is 0.46 times. When running a program or process in per process re-randomization *and* dual process execution (*DE*), we have to include both master and twin into the memory overhead calculation. Therefore, the overhead is calculated by

$$Overhead(ReASLR + DE) = \frac{WS(Twin) + WS(Master)}{WS(Native)} - 1$$

Its overall value is 1.45 times. Note that memory working sets can highly vary during an application's runtime, and thus, are difficult to quantify. The measurements shown in Figure 7 (and in the Tables 5, 6, 7, 8 in Appendix 9.2) were performed after the application has finished startup, and was waiting for user input (i.e., it was idle and all modules were loaded and initialized). Due to additional twins for master processes, the overall additional memory is about one to two times per protected process. The reader is refered to the Appendix 9.2 for more details on data of working set characteristics.

5.1.3 Re-randomization and Startup Time Performance

When a program is started the first time after a reboot, the kernel needs to create section objects for image modules. Hence, the first start of a program always takes longer than subsequent starts of the same program. To measure the additional startup and module load times our protection introduces, we first run each program natively once to allow the kernel to create section objects of most natively used DLLs, and close it afterwards. We then start the program natively without protection and measure the time until it is idle and all of its initial modules are loaded. In the

	Native (8.0)	ReASLR+DE (8.0)	Slowdown	Native (8.1)	ReASLR+DE (8.1)	Slowdown
IE tab spawn	0.9163 s	2.0710 s	1.3x	0.5194 s	1.3082 s	1.5x
Firefox	0.9624 s	1.8064 s	0.9x	1.3823 s	1.5441 s	0.1x
Calculator	0.3484 s	0.3610 s	0.0x	0.4391 s	0.6599 s	0.5x

Table 2: Startup times in seconds and startup slowdowns of native 32-bit applications compared to their counterparts running with per process re-randomization and dual process execution on Windows 8.0 and Windows 8.1 (both 64-bit).

Web page	*google.com*	*facebook.com*	*youtube.com*	*yahoo.com*	*baidu.com*	*wikipedia.org*	*twitter.com*	*qq.com*	*taobao.com*	*linkedin.com*	*amazon.com*	*live.com*	*google.co.in*	*sina.com.cn*	*hao123.com*
Native	425	774	1196	3674	1108	472	599	2405	645	439	958	254	483	3360	373
DETILE	482	961	1519	4722	1339	513	623	2724	824	517	1210	275	517	4269	379
Overhead	13.4%	24.1%	27%	28.5%	20.8%	8.6%	4%	13.2%	27.7%	17.7%	26.3%	8.2%	7%	27%	1.6%

Table 3: Native script execution of IE 11 on Windows 8.1 64-bit compared to the script execution of IE 11 instrumented with DETILE. Execution time is measured in milliseconds using the internal F12 developer tools provided by IE.

same way, we measure the time from process creation until both the master and twin process have their inital modules loaded. The startup comparison can be seen in Table 2. As expected, the startup times of applications protected with our approach are approximately doubled. This is caused by the fact that a twin process needs to be spawned for each master that should be protected.

5.2 Detection Engine

Next, we evaluate the impact of DETILE on the user experience and its effectiveness in detecting information leaks. We performed tests on the script execution time for popular websites and used the prototype to detect four real-world vulnerabilities and a toy example to evaluate our detection capabilities.

5.2.1 Script Execution Time and Responsiveness

We used the 15 most visited websites worldwide [3] to test how the current prototype interferes with the normal usage of these pages. Besides the subjective impression while using the page, we utilized the F12 developer tools of Internet Explorer 11 to measure scripting execution time provided by the *UI Responsiveness* profiler tab. These tests were performed using Windows 8.1 64-bit and Internet Explorer 11. While we introduce a performance hit of around 17.0% on average, the subjective user experience was not noticeably affected. This is due to IE's deferred parsing, which results in displaying content to the user before all computations have finished.

5.2.2 Information Leak Detection

We tested our approach on a pure memory disclosure vulnerability (CVE-2014-6355) which allows illegitimately reading data due to a JPEG parsing flaw in Microsoft's Windows graphics component [89]. It can be used to defeat ASLR by reading leaked stack information back to the attacker via the `toDataURL` method of a `canvas` object. We successfully detected this leak at the point of the call to `toDataURL` in the master and twin process. In the same way, detection was successful for an exploit for a similar bug (CVE-2015-0061 [90]).

To further verify our prototype, we evaluated it against an exploit for CVE-2011-1346, a vulnerability that was used in the pwn2own contest 2011 to bypass ASLR [91]. As this memory

disclosure bug is specific for IE 8, we ported the vulnerability into IE 11. An uninitialized `index` attribute of a new HTML `option` element is used to leak information. Similarly, we successfully detected this exploitation attempt when the `index` attribute was accessed.

Additionally, we tested our prototype on another real-world vulnerability (CVE-2014-0322) that was used in targeted attacks [34] and works for Internet Explorer 10 on Windows 8 64-bit. It is a use-after-free error that can be utilized to increase an arbitrary bit, which is enough to allow read and write access to the complete process memory and create information leaks [50]. Ultimately, exercising the vulnerability allows to read a Typed Array vtable. Thus, it can be accessed and the illegal transition from native memory to the untrusted JavaScript context is performed. Put differently, an information leak is created, which now can be used to reconstruct the complete memory layout. DETILE triggered as the third byte of the Vtable was accessed (i.e., the third least significant byte is the first differing byte in both contexts). Therefore, the information leak was detected successfully without problems.

To further test our implementation, we also constructed a toy example. Thereby, our native code creates an information leak by overwriting the length field of an array. Then, the image base of `jscript9.dll` is written to memory after the array buffer. This ensures that an out of bounds read will result in both, an information leak and a difference in the twin.

We designed the example to be triggered by calling a specifically named JavaScript function and performing array accesses in it. The length field of any arrays accessed in this function will be overwritten with the value `0x400`, allowing memory reads beyond the real array data. This example can be triggered in all versions for which we ported DETILE and only depends on the structure of the internal metadata, which needs to be adjusted between versions of the scripting engine. In our tests, we reliably detected the out of bounds read of the image base and can stop the execution of the process.

5.2.3 False Positive Analysis

We analyzed the 100 top websites worldwide [3] to evaluate if our prototype can precisely handle real-world, complex websites and their JavaScript contexts without triggering false alarms. None of the tested websites did generate an alert, indicating that the prototype can accurately synchronize the master and twin process.

6 Related Work

Software vulnerabilities have received much attention in the last years, mainly due to their high presence in applications and huge impact in practice. Hence, several research results were presented to either offensively abuse vulnerabilities or to develop different defense techniques to mitigate them. In the following, we briefly examine recent work and discuss differences to our approach presented in this paper.

6.1 Security Features Against Memory Corruption Attacks

Modern operation systems apply many methods to harden and protect applications against software vulnerabilities. Amongst others, *Data Execution Prevention* (DEP) [56] forbids executable data, *Address Space Layout Randomization* (ASLR) [65] randomizes the address space of a given program, and SAFESEH/SEHOP provides protection against exception handler manipulation. While DEP and SAFESEH/SEHOP are orthogonal to our system, we improve ASLR on Windows such that each process has its modules rebased to different addresses, turning remote coordinated attacks [51] impractical.

Related to our process-wise re-randomization is Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [54]. While it enforces ASLR for non-ASLR modules, module base addresses are identical in all processes running on a system and change only after a reboot. This still allows coordinated attacks, which EMET cannot protect against. Thus, our re-randomization is

superior as each process has a different module randomization including `kernel32.dll` and the main executable.

6.2 Randomization Techniques

Several approaches have been proposed to either improve address space layout randomization, randomize the data space, or randomize on single instruction level. For example, binary stirring [83] re-randomizes code pages at a high rate for a high performance cost. While it hinders attackers to *use* information leaks in code-reuse attacks, it does not impede their creation by itself. In contrast, our re-randomization scheme reuses the native operating system loader and is the base to allow information leak detection with dual process execution. Oxymoron [5] allows fine-grained address space layout randomization in combination with code sharing and thereby imposing a low overhead. While it protects against code-reuse attacks, it does not *detect* information leaks. A sophisticated attacker is still able to read a complete memory page with sensitive information or manipulate important tokens to escalate privileges. Thus, it does not protect against data-only attacks in combination with information leaks. Our framework differs in that it does not need to rewrite a given binary, and is specialized in determining if memory information is illegally flowing into an untrusted context. Furthermore, we support the 64-bit Windows operating system that is the platform of choice of adversaries to attack applications. Other solutions [47, 61, 61] are prone to JIT-ROP code-reuse attacks [73], which are based on information leaks. Address space layout permutation is an approach to scramble all data and functions of a binary [47]. Therefore, a given ELF binary has to be rewritten and randomization can be applied on each run. ORP [61] rewrites instructions of a given binary and reorders basic blocks. As discussed above, it is prone to information leak attacks, which we detect. Instruction set randomization [6, 46] complicates code-reuse attacks as it encrypts code pages and decrypts it on the fly. However, in the presence of information leaks combined with key guessing [73, 76, 84] it can be circumvented. Instruction layout randomization (ILR) [43] randomizes the location of each instruction on each run, but no re-randomization occurs. Thus, the layout can be reconstructed with the help of an information leak. *Readactor* is a defensive system that aims to be resilient against just-in-time code-reuse attacks [24]. It hides code pointers behind execute-only trampolines and code itself is made execute-only, to prevent an attacker building a code-reuse payload just-in-time. However, it has been shown that it is vulnerable against an attack named *COOP*, which reuses virtual functions [66]. Unlike Readactor, DETILE prevents COOP, as this attack needs an information leak as first step. Crane et al. recently presented an enhanced version of Readactor, dubbed *Readactor++* [25], that also protects against whole function reuse attacks such as COOP. This is achieved through function pointer table randomization and insertion of booby traps. Consequently, an adversary can no longer obtain meaningful code locations that can be leveraged for code-reuse attacks. Readactor++ also does not detect or prevent the exploitation of memory disclosures, which poses a potential attack vector.

6.3 Control-Flow Protections

Web browser have a long history of software vulnerabilities. The most prevalent bugs are use-after-free errors [15], which adversaries are able turn into arbitrary writes nowadays [87, 88]. Hence, protections are arising which specifically target so called vtable-hijacking [18, 36, 63], an exploitation technique used against C++ compiled applications, especially web browsers. The proposed approaches detect and mitigate the usage of fake vtables at virtual function call sites, but cannot protect against the step of creating information leaks, which the adversary needs. Only after bypassing ASLR with an information leak, she is able to gain control over the program at virtual function call sites.

A different approach to mitigate vulnerabilities is to hinder a specific technique of code-reuse attacks, namely *Return-Oriented Programming* (ROP) [20, 35, 62]. Using processor features and heuristics, these works aim to detect so called "gadgets" that attackers utilize in an exploitation attempt during runtime of a program. Overcoming these protections was the focus of recent

research [17, 39, 67], whereby all proposed attacks used information leaks as a first step towards exploitation.

A more general protection against control-flow hijacking attacks is *Control-Flow Integrity* (CFI) [1]. The idea is to instrument a program and verify that each control transfer is jumping to a legitimate control target. Imposing a high performance impact has prevented wide adoption yet. However, coarse-grained CFI protections [93, 94] have a low performance overhead at the expense of security guarantees proposed in the original protection [1]. It has been shown by recent works, that these coarse-grained CFI protections can be circumvented [29, 38]. It is important to notice that these works needed information leaks in the first place to be able to bypass CFI. As the goal of our work is to detect exploits utilizing information leaks, it is orthogonal to CFI and serves as an additional boundary to impede successful exploitation.

6.4 Memory Safety

In a broader sense, monitoring or modifying memory is an approach to mitigate software vulnerabilities. CLING [2] is a memory allocator that only allows memory reuse among objects of the same type, thus, effectively mitigating use-after-free vulnerabilities. It imposes a low performance overhead of 2%. SOFTBOUND [57] guarantees complete memory safety at the cost of a one to two times slowdown and needs access to the source code. As any type of vulnerability can be transformed into an information leak [78], protecting against a specific type of error such as use-after-free is not enough. Also, most applications targeted by sophisticated attackers are available only in binary form, where the source code is not available. Therefore, there is a need of methods, which can handle binary code.

6.5 Multi-Execution Approaches

Most closely related to our research are *n-variant systems*, which run variants of the same program with diverse memory layout and instructions [23]. Bruschi et al. presented a similar work that runs *program replicæ* synchronized at system calls to detect attacks [14, 44]. They demonstrate the detection of memory exploits against the lightweight server `thttpd` on the Linux platform.

Our concept for the detection of information leaks incorporates ideas like dual process execution, per process re-randomization and synchronization. Hence, our work is closely related to n-variant systems [14,23]. However, our approach is adapted specifically for script engines, making it more fine-grained. More specifically, it operates and synchronizes on the bytecode level, whereas n-variant systems intercept system calls.

Furthermore, n-variant systems aim to detect the exploitation of different classes of memory corruption vulnerabilities depending on the utilized diversification method. In contrast, we focus on the *detection* of information leaks, which represent the first step in modern attacks. As such, our approach is capable of identifying the early phase of an attack instead of merely determining that the control flow has diverged.

The major drawback of theses systems is the detection approach: if a memory error is abused, one of the variants eventually crashes, which indicates an attack. As information leaks *do not* constitute a memory error, they *do not* raise any exception-based signal. Thus, they remain undetected in these systems. One significant implication is that unlike DETILE, n-variant systems do not protect against just-in-time code-reuse attacks such as JIT-ROP [73]. Similarily, this is the case with COOP attacks in browsers [66]. N-variant systems prevent conventional ROP attacks [64, 81] with multi process execution and disjunct virtual address spaces: An attacker supplied absolute address (e.g., obtained through a remote memory disclosure vulnerability) is guaranteed to be invalid in $n-1$ replicas. Hence, any system call utilizing this address will trigger a detection. However, JIT-ROP attacks may performs several memory disclosures and malicious computations without executing a system call inbetween, and thus, can evade traditional n-variant systems. COOP attacks may as well perform touring-complete computations on disclosed memory without executing a system call and evade these systems.

Additionally, we show that synchronized interpreter execution can be achieved in much more complex software systems with much higher synchronization granularity, even without having access to the source code.

Private information leaks can be prevented with *shadow executions* [16]. In contrast, our prototype does not require a virtual machine per program to perform multi execution as our prototype achieves synchronization on the binary level. Additionally, we aim to detect general and fine-grained raw memory information leaks usable to bypass security features. Other research showed that private information leak in networks can be prevented via comparing outbound traffic of original and shadow processes for divergence [26]. Our work strongly varies, as we focus on inherent binary flows and show that even in highly complex software, subtle differences can be used to detect illegal program behavior.

Our dual execution approach differs to other multi-execution approaches in that *both* programs *do not* need to receive input tagged at a security level. This is due to the fact that raw memory leaks show themselves as different contents inside low privileged contexts.

DieHard [8] and DieHarder [58] are memory allocators that mitigate heap vulnerabilities. Furthermore, DieHard is able to operate in a so called *replicated mode* to run a program several times in parallel to compare the output. In spirit, this is similar to our approach. However, the design of DieHard does not allow to run programs which perform network operations or access the filesystem. In our work, we show that multi-execution is possible beyond programs in the scope of DieHard, e.g., in sophisticated and complex software like web browsers without having access to their source code. Additionally, in single execution mode, memory disclosures like uninitialized reads fly under the radar of DieHard and DieHarder as they do not constitute necessarily a memory error.

Devriese and Piessens showed that *noninterference* can be achieved via *secure multi-execution* [31]. It targets a different context than our work and is implemented against source code to compare only JavaScript I/O. Similarly, it is possible to leverage static transformation of JavaScript and Python code of interest to obtain secure multi-execution [7]. Our work can serve as a starting point to achieve non-interference in binary programs which encompass security levels spread over difficult to connect boundaries (i.e., native memory and the JavaScript layer).

7 Discussion

In the following, we discuss potential shortcomings of our approach and the prototype, and also sketch how these shortcomings can be addressed in the future.

7.1 Further Information Leaks

Serna provided an in-depth overview of techniques that utilize information leaks for exploit development [71]. The techniques he discussed during the presentation utilize JavaScript code. As our prototype leverages the JavaScript engine of the browser itself, each information leak that is based on these techniques is detected. This implies that memory disclosure attacks that leverage other (scripting) contexts (e.g., VBScript) can potentially bypass our implementation. However, in practice exploits are typically triggered via JavaScript and thus our prototype can detect such attacks. Furthermore, due to the generic nature of our approach, our current prototype can be extended by instrumenting other scripting engines as well. Another flavor of information leak is based on timing attacks. In 2012, it has been shown that timing attacks on the hash table implementation in Firefox can be utilized to obtain addresses of string objects [60]. In 2013, another timing attack was presented that uses the garbage collector to leak addresses of internal data [11]. Hund et al. demonstrate that kernel-space ASLR can be defeated by timing attacks targeted at the cache and TLB [45]. Seibert et al. demonstrate how memory corruption vulnerabilities can aid in utilizing timing side channel techniques to gain knowledge about the executed code, even if the code is diversified [69]. Timing based information leaks are difficult to detect, as it is hard to verify if a computational operation serves the purpose of a timing side channel or if it is

legitimate. Hence, the reliable detection of information leaks based on timing side channels has yet to be shown, but recent results are promising [32, 41]. While our detection does not trigger on timing-based information leaks, it triggers on the *usage* of them, because we monitor the transition into native memory from JavaScript. As such, we cannot directly observe the information leak, but can directly detect its results once it enters the scripting context or flows from the scripting context to native memory.

7.2 Limitations of Prototype Implementation

In the unlikely event one of the functions we classified as entropy source, such as `Math.random` or `Date.now`, contain a memory disclosure bug, our approach can lead to an under-approximation of detected information leaks. In this specific case, the master confuses the leaked pointer with data from the entropy source and transfers it to the twin process. This is an undesirable state, because DETILE does not prevent the memory layout information to leak into the script context. However, the obtained pointer is only valid in the master process. An attempt to leverage the pointer to mount a code-reuse attack crashes the twin. As a consequence, DETILE halts the master process and prevents further damage.

The current prototype disables the JIT Engine as we protect the interpreter only. However, *dynamic binary instrumentation* (DBI [13, 53]) frameworks allow to synchronize processes on the intruction or basic block level, and hence, make it possible to hook emitted JIT code to dispatch our assembly stub in order to synchronize and check within the JIT code.

Asynchronious JavaScript events are currently not synchronized. This is solvable with DBI frameworks as well: If an event triggers in the master process, we let the twin execute to the same point. Then DETILE sets up and triggers the same event in the twin process.

One additional shortcoming of our prototype implementation is the identical mapping of `ntdll.dll` in all processes. As this DLL is initialized already at startup, remapping it is a cumbersome operation. JavaScript, HTML, and other contexts in browsers normally do not interact directly with native `ntdll.dll` Windows structures, and thus internal JavaScript objects, do not contain direct memory references to it. Hence, attackers resort to disclose addresses from libraries other than `ntdll.dll` at first. On the contrary, there might be script engines which directly interact with `ntdll.dll`. Still, the issue is probably solvable with a driver loaded during boot time.

Another technical drawback is the application of re-randomization on every process on the OS, as DLL modules of each process would turn into non-shareable memory and increase physical memory consumption. This can be avoided by protecting only critical processes that represent a valid target for attacks.

7.3 Deployment

The current prototype is not meant to be a protection framework for end users of web browsers. It is intended to be deployed as a system for scanning web pages to discover unknown exploits which utilize information leaks. As ASLR needs to be circumvented as a first step of each modern exploit against web browsers, DETILE has the advantage to provide an early detection of the exploit process.

8 Conclusion

Over the last years, script engines were used to exploit vulnerable applications. Especially web browsers became an attractive target for a plethora of attacks. State-of-the-art vulnerability exploits, both in academic research [17,29,38,39,67] and in-the-wild [68,82,86–88], rely on memory disclosure attacks.

In this work, we proposed a fine-grained, automated scheme to reliably detect such information leaks in script engines. It is based on the insight that information leaks result in a noticable

difference in the script context of two synchronized processes with different randomization. We implemented a prototype of this idea for the proprietary browser IE to demonstrate that our approach is viable even on closed-source systems. An empirical evaluation demonstrates that we can reliably detect real-world attack vectors and that the approach induces a moderate performance overhead only (around 17% overhead on average). While most research focused on mitigating specific types of vulnerabilities, we address the root cause behind modern attacks since most of them rely on information leaks as a first step. Our approach thus serves as another defense layer to complement defenses such as DEP and ASLR.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the European Commission through the ERC Starting Grant No. 640110 (BASTION).

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

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

[3] Alexa. The top 500 sites on the web. `http://www.alexa.com/topsites`, 2014.

[4] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser JIT engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[5] M. Backes and S. Nürnberger. Oxymoron: making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

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

[7] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. pages 186–202, 2012.

[8] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy*, 2014.

[10] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *BlackHat DC*, 2010.

[11] D. Blazakis. GC woah. `http://www.trapbit.com/talks/Summerc0n2013-GCWoah.pdf`, 2013.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[13] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[14] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicæ for defeating memory error exploits. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, 2007.

[15] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[16] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[17] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[18] Z. C. Chao Zhang Chengyu Song, Kevin Zhijie Chen and D. Song. VTint: Defending Virtual Function Tables Integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[19] 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 (CCS)*, 2010.

[20] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[21] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

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

[23] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant Systems: A Secretless Framework for Security Through Diversity. In *USENIX Security Symposium*, 2006.

[24] 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*, 2015.

[25] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRAP: Table randomization and protection against function reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[26] J. Croft and M. Caesar. Towards practical avoidance of information leakage in enterprise networks. In *HotSec*, 2011.

[27] CVE. Google Chrome Vulnerability Statistics. `http://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224`, 2014.

[28] CVE. Microsoft Internet Explorer Vulnerability Statistics. `http://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html?vendor_id=26`, 2014.

[29] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[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 (NDSS)*, 2015.

[31] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, 2010.

[32] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*, 2013.

[33] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Symposium on Operating Systems Design and Implementation*

(OSDI), 2006.

[34] FireEye. Operation SnowMan. http://www.fireeye.com/blog/technical/cyber-exploits/2014/02/operation-snowman-deputydog-actor-compromises-us-veterans-of-foreign-wars-website.html, 2014.

[35] I. Fratric. Runtime Prevention of Return-Oriented Programming Attacks. http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf.

[36] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[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 (NDSS)*, 2016.

[38] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[39] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium*, 2014.

[40] D. Herman and K. Russell. Typed array specification. *Khronos. org*, 2011.

[41] J. Heusser and P. Malacaria. Quantifying Information Leaks in Software. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[42] T. Hirvonen. Dynamic flash instrumentation for fun and profit. *Black Hat USA*, 2014.

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

[44] P. Hosek and C. Cadar. Varan the unbelievable: An efficient n-version execution framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[45] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*, 2013.

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

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

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

[49] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[50] B. Labs. Dissecting the newest IE10 0-day exploit (CVE-2014-0322). http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/, 2014.

[51] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened ASLR on Android. In *IEEE Symposium on Security and Privacy*, 2014.

[52] H. Li. Inside AVM. In *REcon*, 2012.

[53] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm Sigplan Notices*, 2005.

[54] Microsoft. EMET 5.1 is available. `http://blogs.technet.com/b/srd/archive/2014/11/10/emet-5-1-is-available.asp`, 2014.

[55] Microsoft. What is the Windows Integrity Mechanism? `http://msdn.microsoft.com/en-us/library/bb625957.aspx`, 2014.

[56] I. Molnar. Exec shield, new linux security feature. *News-Forge, May*, 2003.

[57] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, 2009.

[58] G. Novark and E. D. Berger. DieHarder: securing the heap. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

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

[60] pakt. Leaking information with timing attacks on hashtables, 2012.

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

[62] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security Symposium*, 2013.

[63] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[64] M. Prandini and M. Ramilli. Return-Oriented Programming. In *IEEE Symposium on Security and Privacy*, 2012.

[65] M. Russinovich, D. Solomon, and A. Ionescu. *Windows Internals, Part 2*. Microsoft Press, 2012.

[66] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming. In *IEEE Symposium on Security and Privacy*, 2015.

[67] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2014.

[68] V. Security. Advanced Exploitation of Mozilla Firefox Use-after-free (MFSA 2012-22). `http://www.vupen.com/blog/20120625.Advanced_Exploitation_of_Mozilla_Firefox_UaF_CVE-2012-0469.php`, 2012.

[69] J. Seibert, H. Okkhravi, and E. Söderström. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[70] F. J. Serna. CVE-2012-0769, the case of the perfect info leak. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

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

[72] 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 (CCS)*, 2007.

[73] 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*, 2013.

[74] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *IEEE Symposium on Security and Privacy*, 2016.

[75] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[76] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the effectiveness of instruction set

randomization. In *USENIX Security Symposium*, 2005.

[77] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *ACM European Workshop on System Security (EUROSEC)*, 2009.

[78] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*, 2013.

[79] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[80] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[81] S. Volckaert, B. Coppens, and B. De Sutter. Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[82] P. Vreugdenhil. A browser is only as strong as its weakest byte - Part 2. http://blog.exodusintel.com/2013/12/09/a-browser-is-only-as-strong-as-its-weakest-byte-part-2/, 2012.

[83] 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 (CCS)*, 2012.

[84] Y. Weiss and E. G. Barrantes. Known/chosen key attacks against software instruction set randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[85] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. 2016.

[86] T. Yan. The art of leaks: The return of heap feng shui. In *CanSecWest*, 2014.

[87] Y. Yu. ROPs are for the 99%. In *CanSecWest*, 2014.

[88] Y. Yu. Write Once, Pwn Anywhere. In *Black Hat USA*, 2014.

[89] M. Zalewski. Two more browser memory disclosure bugs. `http://lcamtuf.blogspot.de/2014/10/two-more-browser-memory-disclosure-bugs.html`, 2014.

[90] M. Zalewski. Bi-level TIFFs and the tale of the unexpectedly early patch. `http://lcamtuf.blogspot.de/2015/02/bi-level-tiffs-and-tale-of-unexpectedly.html`, 2015.

[91] ZDI. CVE-2011-1346, (Pwn2Own) Microsoft Internet Explorer Uninitialized Variable Information Leak Vulnerability. `http://www.zerodayinitiative.com/advisories/ZDI-11-198/`.

[92] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE). `http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx`, 2008.

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

[94] M. Zhang and R. Sekar. BinCFI: Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, 2013.

9 Appendix

9.1 Re-randomized Process Modules

DLL Name	Systemwide	Re-Randomized Process 1	Re-Randomized Process 2
ntdll.dll (64-bit)	0x7FF9D5DD0000	0x7FF9D5DD0000	0x7FF9D5DD0000
ntdll.dll	0x77640000	0x77640000	0x77640000
wow64win.dll (64-bit)	0x775D0000	0x590000	0xCD0000
wow64.dll (64-bit)	0x77580000	0x290000	0xC80000
wow64cpu.dll (64-bit)	0x77570000	0xA0000	0x8C0000
oleaut32.dll	0x773A0000	0x2770000	0x2310000
advapi32.dll	0x77270000	0x2A40000	0x2620000
msvcrt.dll	0x76E80000	0xB80000	0x12E0000
kernel32.dll	0x76AA0000	0x750000	0xE10000
ws2_32.dll	0x76A50000	0x56D0000	0x5DD0000
KernelBase.dll	0x76840000	0x890000	0x1070000
gdi32.dll	0x76700000	0x1330000	0x1A50000
shlwapi.dll	0x76620000	0x2610000	0x21B0000
user32.dll	0x764D0000	0xE60000	0x15C0000
crypt32.dll	0x76350000	0x8520000	0x8BA0000
ole32.dll	0x76240000	0x2660000	0x2200000
shell32.dll	0x75080000	0x3EF0000	0x4A40000
cryptbase.dll	0x75050000	0x270000	0xC70000
apphelp.dll	0x74F50000	0xAE0000	0x1240000
ieframe.dll	0x74160000	0x1AC0000	0x3EF0000
IEShims.dll	0x73F20000	0x29F0000	0x25D0000
wininet.dll	0x73C60000	0x53D0000	0x5C00000
userenv.dll	0x73C40000	0x5330000	0x2930000
urlmon.dll	0x73AF0000	0x55A0000	0x2990000
winhttp.dll	0x73A40000	0x5720000	0x5E20000
mswsock.dll	0x739F0000	0x5B70000	0x6280000
rsaenh.dll	0x73970000	0x5C30000	0x6370000
ieproxy.dll	0x738E0000	0x5F90000	0x66D0000
dnsapi.dll	0x73820000	0x6380000	0x8080000
mshtml.dll	0x72340000	0x648000	0x6980000
d2d1.dll	0x71F70000	0x76A0000	0x7C40000
ieui.dll	0x71D10000	0x7F10000	0x8530000
jscript9.dll	0x718B0000	0x8110000	0x8770000
d3d11.dll	0x71630000	0x8FD0000	0x7990000
iexplore.exe	0x2F0000	0x3C0000	0xA70000

Table 4: Re-randomized processes in Internet Explorer 11: original processes have their modules mapped systemwide at same base addresses, while our re-randomized processes map their modules to a different base processwise. Bold entries represent the system's and the browser's most essential modules.

9.2 Working set characteristics of per process re-randomization and dual process execution

Table 5 and Table 7 show the native memory consumption of three applications whose consumptions are shown in Table 6 and Table 8 when running in dual process execution mode. As Internet Explorer is a multi-process software (see Section 2.4), tabs can run in separate processes. Therefore, there exist one 64-bit main IE frame and one tab process in native mode, while one 64-bit IE frame manages an additional twin process per master tab process when running in dual process execution mode. The results show, what intuitively is clear: As a re-randomized process has its modules loaded on different base addresses than other processes, corresponding memory is not

Process	Working Set	WS Private	WS Shareable	WS Shared
iexplore.exe (64-bit frame)	20,980 K	3,008 K	17,972 K	14,532 K
iexplore.exe	31,796 K	5,260 K	26,536 K	10,888 K
firefox.exe	87,856 K	47,988 K	39,868 K	16,320 K
calc.exe	11,440 K	1,332 K	10,108 K	8,968 K

Table 5: Memory working sets of 32-bit native processes running on Windows 8.1 64-Bit (Internet Explorer 11 main frame with one tab, Mozilla Firefox 33.0 and the Windows Calculator)

Process	Working Set	WS Private	WS Shareable	WS Shared
iexplore.exe (64-bit frame)	30,516 K	7,480 K	23,036 K	17,980 K
iexplore.exe (master)	26,204 K	5,008 K	21,196 K	18,592 K
iexplore.exe (twin)	51,816 K	43,476 K	8,340 K	8,332 K
iexplore.exe (master)	21,556 K	3,928 K	17,628 K	17,536 K
iexplore.exe (twin)	52,004 K	43,596 K	8,408 K	8,368 K
firefox.exe (master)	88,936 K	53,388 K	35,548 K	16,512 K
firefox.exe (twin)	95,516 K	80,316 K	15,200 K	14,208 K
calc.exe (master)	11,828 K	4,880 K	6,948 K	6,180 K
calc.exe (twin)	20,748 K	14,784 K	5,964 K	5,912 K

Table 6: Memory Working Sets of 32-bit processes running in dual execution mode on Windows 8.1 64-bit. All twin processes are differently randomized (Internet Explorer 11 main frame with two tabs, Mozilla Firefox 33.0 and the Windows Calculator)

Process	Working Set	WS Private	WS Shareable	WS Shared
iexplore.exe (64-bit frame)	19,996 K	4,264 K	15,732 K	13,456 K
iexplore.exe	31,432 K	7,580 K	23,852 K	9,840 K
firefox.exe	83,988 K	52,400 K	31,588 K	10,684 K
calc.exe	11,316 K	1,632 K	9,684 K	9,052 K

Table 7: Memory Working Sets of 32-bit native processes running on Windows 8.0 64-Bit (Internet Explorer 10, Firefox, and the Calculator)

Process	Working Set	WS Private	WS Shareable	WS Shared
iexplore.exe (64-bit frame)	27,132 K	7,160 K	19,972 K	16,824 K
iexplore.exe (master)	30,860 K	7,736 K	23,124 K	19,184 K
iexplore.exe (twin)	55,112 K	39,196 K	15,916 K	14,788 K
iexplore.exe (master)	27,108 K	5,380 K	21,728 K	18,220 K
iexplore.exe (twin)	55,368 K	39,464 K	15,904 K	14,780 K
firefox.exe (master)	85,968 K	51,240 K	34,728 K	18,120 K
firefox.exe (twin)	99,740 K	82,488 K	17,252 K	15,876 K
calc.exe (master)	11,808 K	5,020 K	6,788 K	6,376 K
calc.exe (twin)	14,840 K	9,052 K	5,788 K	5,724 K

Table 8: Memory Working Sets of 32-bit processes running in dual execution mode on Windows 8.0 64-bit with all twin processes differently randomized (Internet Explorer 10, Firefox, and the Calculator)

shared. The process needs private copies for its modules. This results in higher working sets, lower shareable and lower shared memory in re-randomized processes (twins) compared to their master processes.

9.3 Internals of information leak exploit for CVE-2014-0322

Recently, attackers began to use techniques [86, 88] to gain access to the process' virtual address space from JavaScript by changing a single bit only. Thus, it is important to have a good understanding of them to reliably develop mitigations. The exploit works in the following way: the heap is shaped in a specific layout with heap feng shui [86], such that general arrays are aligned to `0xXXXX0000` boundaries and headers of Typed Arrays follow aligned at `0xXXXXf000`. The structure after heap feng shui is performed is illustrated in Figure 8.

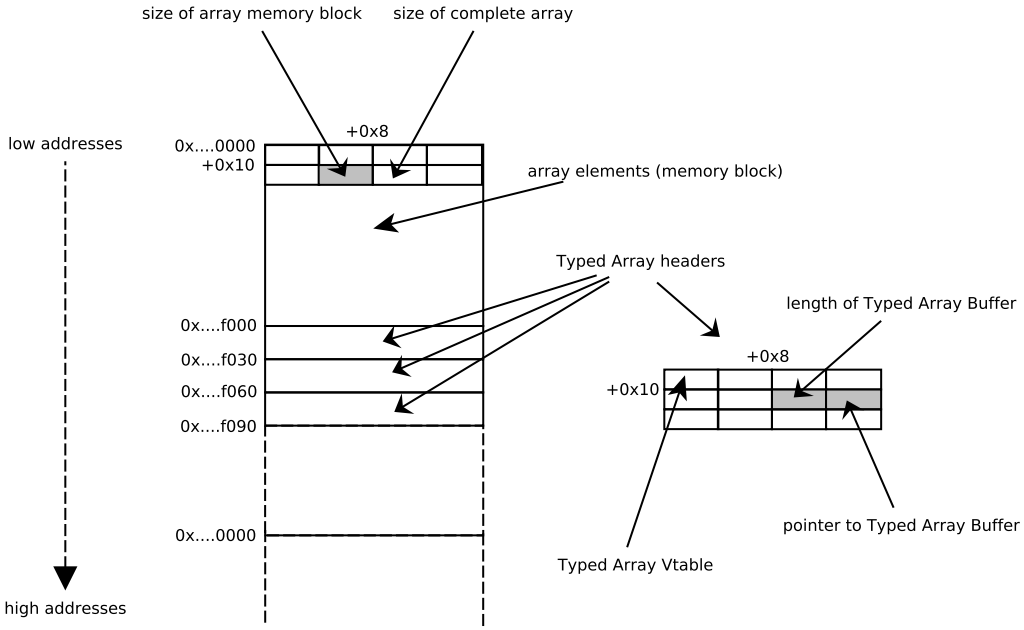

Figure 8: Generating an information leak with CVE-2014-0322: Modified fields are shaded gray. The vulnerability allows a bit increase in the size of the array memory block. This is sufficient to subsequently and illegitimately change the length of a Typed Array Buffer and the pointer to a Typed Array Buffer in the contiguous Typed Array Header. This results in access to the complete process memory.

The vulnerability is used to increase the most significant byte of the size of the array memory block. This is possible, as a function operates on the injected fake object data with the code `inc [eax + 0x10]`. Then, we can perform out of bound writes with JavaScript methods of the modified array. We first change the pointer to a Typed Array Buffer in the Typed Array header following the general array data. We then, change also the length of the Typed Array Buffer to `0x7fffffff` in the header. Now we have access to the process' virtual address space via JavaScript methods of the changed Typed Array. This allows leaking any memory such as vtable pointers or code.