Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding

Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany and Thorsten Holz
Horst Görtz Institute for IT-Security (HGI), Ruhr-University Bochum, Germany
{firstname}.{lastname}@rub.de

***Abstract*—It is a well-known issue that attack primitives which exploit memory corruption vulnerabilities can abuse the ability of processes to automatically restart upon termination. For example, network services like FTP and HTTP servers are typically restarted in case a crash happens and this can be used to defeat *Address Space Layout Randomization* (ASLR). Furthermore, recently several techniques evolved that enable complete process memory scanning or code-reuse attacks against diversified and unknown binaries based on automated restarts of server applications. Until now, it is believed that client applications are immune against exploit primitives utilizing crashes. Due to their hard crash policy, such applications do not restart after memory corruption faults, making it impossible to touch memory more than once with wrong permissions.**

In this paper, we show that certain client application can actually *survive* crashes and *are able* to tolerate faults, which are normally critical and force program termination. To this end, we introduce a *crash-resistance* primitive and develop a novel memory scanning method with *memory oracles* without the need for control-flow hijacking. We show the practicability of our methods for 32-bit Internet Explorer 11 on Windows 8.1, and Mozilla Firefox 64-bit (Windows 8.1 and Linux 3.17.1). Furthermore, we demonstrate the advantages an attacker gains to overcome recent code-reuse defenses. Latest advances propose fine-grained re-randomization of the address space and code layout, or hide sensitive information such as code pointers to thwart tampering or misuse. We show that these defenses need improvements since crash-resistance weakens their security assumptions. To this end, we introduce the concept of *Crash-Resistant Oriented Programming* (CROP). We believe that our results and the implications of memory oracles will contribute to future research on defensive schemes against code-reuse attacks.

I. Introduction

In the last years, attackers shifted their focus away from network services to client applications and especially web browsers became an attractive target. Adversaries can apparently easily detect memory corruption and similar vulnerabilities in such complex programs, as demonstrated by the steady stream of reported vulnerabilities. In contrast to server software, client applications have a crucial property: they typically terminate immediately on memory corruption faults or on failed exploit attempts, and do not automatically restart themselves. Hence, adversaries have usually only one shot to conduct their attack successfully. In contrast, the ability of network services to restart initiated research on developing sophisticated attack primitives: if programs such as servers automatically respawn after termination due to a crash, memory layout information or hidden sections can be deduced which are otherwise not accessible to an adversary [6, 19, 50]. Until now, a common belief is that such attacks can only be conducted against restarting programs, especially network services like FTP and HTTP servers.

In this paper, we challenge this assumption and demonstrate the ability to *handle faults* in a manner that memory corruptions no longer remain an all or nothing primitive against client software. More specifically, we demonstrate that memory corruption vulnerabilities can in fact be used as a side channel to weaken available security features.

A. Defenses Against Memory Corruption Attacks

For some years, we observe an ongoing arms race between attackers and defenders in the area of memory corruption vulnerabilities. From time to time, novel attacks are proposed that break existing defenses and it is still an open research problem how memory corruption vulnerabilities can be prevented with a reasonable performance overhead. Gradually it became apparent that there are several building blocks that can be used on the defense side to hamper attacks. First, there is the $W \oplus X$ (**W**ritable **xor** e**X**ecutable) security model [39] that prevents an attacker from redirecting the hijacked control flow to data of her choice that is then interpreted as code. The $W \oplus X$ security model is nowadays directly supported within processors and the Windows operating systems supports *Data Execution Prevention* (DEP) since 2004. In response, different kinds of code-reuse attacks such as for example *return-to-libc*, *Return-Oriented Programming* (ROP) [49], and many more variants [7, 12, 44] were developed. Recently, many detection techniques for code-reuse attacks were proposed [20, 36], but most of them were also broken shortly afterwards [10, 22, 47].

A second building block is randomization of the address space, a strategy which can be leveraged to prohibit an attacker from re-using existing code since the location of code (or data structures) is not known beforehand. Many kinds of randomization strategies were proposed over the years [4, 24, 38] that hide the code layout, and *Address Space Layout Randomization* (ASLR) [38] is widely deployed on modern operating systems. It is a well-known issue that the low entropy used to randomize a program's address space

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23262

on 32-bit systems is susceptible to brute-force attacks [50]. However, discovering parts of the memory layout in a brute force manner requires a program which tolerates crashes. Thus, this way of exploitation is normally only viable on certain kinds of server software, where each request spawns a new but equally randomized process. With increasing entropy to randomize the address space on 64-bit architectures and a hard crash policy which forbids restarts of the program, brute-force attacks do not seem to be a viable option anymore these days. In addition, efficient (re-)randomization schemes seem to be a promising direction and several papers that propose such randomization schemes were recently published [3, 13, 17]. It remains an open question if attacks against such schemes are viable, especially in the light of memory disclosure attacks.

A third building block attempts to enforce *Control-Flow Integrity* (CFI) [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 [18, 26, 59, 60]. However, several papers recently demonstrated bypasses of these CFI solutions [16, 19, 21].

B. Subverting Information Hiding

In the presence of these defenses, the successful exploitation of a memory corruption vulnerability poses a challenge to an adversary. We observe that a successful attack is typically based on a memory disclosure vulnerability (so called *information leak*): such leaks are often the first step utilized to gain some knowledge about the memory layout. Once the code locations are collected, they can be used to mount a code-reuse attack (which is in turn used to disable $W \oplus X$).

More importantly, we observe that hiding of information in a program's address space is a crucial aspect: data structures with sensitive information have to remain hidden in order to prevent weakening available security features. For example, in the presence of ASLR, base addresses of shared modules as well as stack addresses of running threads, heap boundaries, and exception handlers have to remain hidden from an attacker. With fine-grained randomization schemes [5, 24, 56], the same problem applies: an adversary might be able to uncover the address space via novel attack methods and leverage this information to perform just-in-time attacks similar to the work by Snow et al. [51].

Other security features rely on information hiding as well, for example hidden regions to store metadata used to perform integrity checks. Consider for example *Code-Pointer Integrity* (CPI) [26], the state-of-the-art code pointer protection approach: on platforms where the implementation is based on information hiding (e.g., Intel's x86-64 architecture), such pointers (and pointers to such pointers) are stored in a hidden memory region to impede tampering with them. Recently and concurrently to our work, a successful attack against the current CPI implementation was demonstrated that leaks this hidden memory region [19]. Furthermore, all CFI implementations that leverage a shadow stack need to prevent this stack from being leaked to an attacker [9, 15].

C. Novel Memory Probing Method

In this paper, we show that fault-tolerant functionality is available in web browsers and—when combined with memory disclosures—delivers a novel way to explore unknown memory territories. It is a common belief that a hidden memory region without references to it is undiscoverable in practice without code-execution or code-reuse attacks. Thus, an important building block towards revealing reference-less memory is the ability to scan the address space *without* forcing the program into termination. This is viable in server software [6, 19, 50], but seems impossible in web browsers due to their hard crash policy (i.e., after three consecutive crashes, Internet Explorer stops restarting automatically). We demonstrate that a memory scanning ability in web browsers can be achieved and use this as a base to subvert memory secrecy and randomization approaches *without* control-flow hijacking, code injection or code-reuse attacks. Deducing hidden information with memory scans *in turn* enables an adversary to conduct code-reuse attacks.

In our experiments, we were able to scan the address space with 18,357 probes per second in 64-bit Firefox on Linux, with 718 probes per second in 64-bit Firefox on Windows, and 63 probes per seconds in 32-bit Internet Explorer. We leverage *memory oracles* as an extension of information leaks, which either return the content at specified memory or deliver an event in case of unmapped memory, to learn more about the structure of the address space. This enables us to circumvent standard ASLR implementations and recently proposed defense schemes are undermined. Additionally, we use our crash-resistance primitive together with function chaining to achieve *Crash-Resistant Oriented Programming* (CROP): arbitrary exported system calls or functions can be dispatched in a fault-tolerant manner.

In summary, we make the following contributions:

- To the best of our knowledge, we are the first to introduce the ability in web browsers to *survive* crashes and to run in fault-tolerant mode. We term this new class of primitive *crash-resistance*.
- We develop new methods allowing to scan memory inside client software based on *crash-resistance* and *memory oracles*. We thereby do not need control-flow hijacking, code injection, or code-reuse. Furthermore, we demonstrate the practical feasibility of our methodology for Internet Explorer 32-bit on Windows and Mozilla Firefox 64-bit on Linux and Windows.
- We present the advantages an adversary gains with crash-resistance and memory oracles to weaken recently proposed security features based on code hiding and (re-)randomization. More specifically, we show that memory secrecy enforced through memory layout randomization is ineffective *even* in a large address space (i.e., on x86-64 systems), uncovering sensitive information protected via information hiding.
- Finally, we develop a new code-reuse technique based on function chaining in combination with crash-resistance. We term this technique *Crash-Resistant Oriented Programming* (CROP).

II. Technical Background

In the following, we first introduce the adversarial capabilities and the defense model we use throughout this paper. Furthermore, we describe several state-of-the-art defense strategies and briefly discuss potential shortcomings.

A. Adversary Model

We assume that the adversary has an initial vulnerability such as a use-after-free or a restricted write (such as a byte increment/decrement or null byte write) to an attacker-chosen address. We assume as well that the initial vulnerability leads to the ability to read from and write to arbitrary addresses, using a scripting environment such as JavaScript. These assumptions are consistent with recent exploitation of memory corruptions in web browsers [13, 14, 17]

Furthermore, we assume that the target system incorporates the following defense mechanisms against the exploitation of memory corruption vulnerabilities:

- *Non-executable memory*: The target OS implements the $W \oplus X$ security model that is applied to all *non-executable* pages. Thus, only code pages gain the execute permission in order to hamper code injection attacks.
- *Memory diversification*: The adversary has to tackle several levels of randomization techniques starting with the widely deployed coarse-grained ASLR that randomizes modules, over to fine-grained address space randomization on the instruction/basic block/function level [5, 23, 24, 37, 56], to re-randomizing code such as proposed in *Isomeron* [17].
- *Control-flow integrity*: As CFI implementations such as Microsoft's *Control Flow Guard* (MS-CFG) begin to be deployed in commodity operating systems, we assume that coarse-grained CFI [59, 60] and ROP protections [20] are active on the target OS.
- *Execute-no-read memory*: We further restrict the attacker by enforcing non-readable code pages ($R \oplus X$) as proposed by recent JIT-ROP defenses [2, 13]. Consequently, any read attempt to code results in an access fault.
- *Hard crash policy*: We assume that a program does not automatically restart after a crash and that a user will not open a potentially dangerous web page again after it crashed a web browser once.

B. Randomization Techniques

Several randomization techniques were proposed over the last years and we briefly review the different approaches. Furthermore, we discuss potential shortcomings of such methods.

1) Address Space Layout Randomization: All state-of-the-art operating systems deploy *ASLR*. This feature randomizes the base address of shared libraries and executables, all stacks, heaps, and other structures. Randomization is performed at load time of a program and ideally no locations of specific memory are predictable. However, there exist drawbacks: offsets to data structures and code within a shared library remain constant and are susceptible to static code-reuse attacks. If one library base address is revealed with one memory disclosure, the adversary knows the layout of the complete module [53].

2) Fine-Grained ASLR: To overcome the constant layout in shared modules and to prevent an attacker to conduct static code-reuse attacks, several schemes of fine-grained randomization were developed. They randomize the code layout [23], replace instructions with semantic equivalents [37], or alternate the order of basic blocks [56]. These methods are applied during load time of a program. Unfortunately, these defenses can be bypassed if an adversary discloses code pages and assembles a code-reuse payload dynamically on the fly [51].

3) Re-Randomization: To hinder dynamic code-reuse attacks, re-randomization is applied to programs: if an adversary discovers code locations via memory disclosure vulnerabilities, she cannot use them as re-randomization changes the code layout in between. *Isomeron* [17] applies fine-grained randomization in the load phase of a program to ensure that not only basic blocks or modules are placed at different addresses, but also single code snippets. Furthermore, *Isomeron* applies re-randomization on the granularity of function calls during runtime. In a coin-flip manner, it decides whether the original or a diversified copy of a function is executed. This approach thwarts code-reuse attacks like ROP and JIT-ROP. However, we found that specific structures are very challenging to re-randomize, especially data structures to which dynamic access needs to be maintained during a program's runtime (see Section IV-A1). Thus, we show that an adversary can still gather sufficient information and conduct code-reuse attacks.

C. Security by Information Hiding

As noted above, hiding of information in a program's address space is getting more and more into the spotlight of interest. Note that all structures with sensitive information have to remain hidden to prevent an adversary from leaking them (and thus weakening available security features). In the context of ASLR, the following information is for example considered to be sensitive: base addresses of shared modules, stack addresses of running threads, heap boundaries, and exception handlers. Based on information hiding, memory regions without references to them exist for similar reasons: they are based on the assumption that memory disclosures cannot reveal them, as knowledge about their location is not available and they are *reference-less*. We explain several instances of information hiding in the following.

1) Sensitive Application Structures: Microsoft Windows maintains a *Process Environment Block* (PEB) for each running process. Similarly, a *Thread Environment Block* (TEB) is included in each process' address space for each thread. The legitimate method to gain access to either of them is to use the official Windows API call `NtCurrentTeb()`. Accessing a TEB or the PEB illegitimately is often done by using the `FS` register on x86 architectures: the address of the currently active thread's TEB is found at `FS:0` and the address of the PEB at `[FS:0x30]`. To the best of our knowledge, references to both structures do not exist anywhere in user-space memory.

In presence of ASLR, it is nearly impossible for an adversary to reveal the structures with complete read access to memory, unless prior knowledge of the memory layout is available to her. Trying to read unreadable memory results in access faults and termination of the program. For an adversary, the only possible way to reveal them is to hijack the control-flow and execute her code of choice. In Section III-E, we show

that this hidden information is accessible even if code-reuse attacks are not an option (e.g., due to control-flow integrity).

Valuable information for an attacker in a TEB is the thread's stack boundaries or the chain of exception handlers. Very critical is an undocumented code-trampoline field at offset `0xC0`: every system call of a 32-bit process running on 64-bit Windows is going through this CPU-modeswitch trampoline. If an attacker manages to overwrite that field, she can gain control over every system call in that process.

Among other information, the PEB contains the base addresses of all mapped modules and a function callback table for the Windows kernel. While disclosing a module's base address by reading another module's *import address table* (IAT) may be an option, reading the PEB directly yields all executable modules of a process at once. Note that there are no references to kernel callback tables in the user-mode address space for obvious reasons.

2) Reference-Less Regions for Pointer Safety: Another notable example of this concept are implementations of *Code-Pointer Integrity* (CPI [26]) on x86-64 and ARM systems. In absence of hardware enforced segmentation protection, the CPI implementation on these platforms relies on hiding the location of the *safe region* that contains the sensitive metadata for pointers. The safe region is used by CPI to enforce its policies. The most restrictive variant of the CPI policy tracks all sensitive pointers in a program. A pointer is considered sensitive if it is a code pointer or a pointer that may later be used to access a sensitive pointer. This recursive definition ensures that all control flow information is protected.

The security of CPI on the x86-64 and ARM architectures relies on hiding the precise location of metadata from an attacker. This concept has already been shown to be susceptible to attacks by Evans et al. [19]. In Section III-E, we present an even more efficient mechanism to determine the location of hidden memory. It is used to launch a similar attack on CPI *without* crashes in a shorter time.

3) Code and Code Pointer Hiding: In case fine-grained randomization is in place, an adversary can still conduct JIT-ROP attacks [51]. To prevent the attacker's ability to discover enough code to reuse, recent research has focused on mapping code as execute-only [2] or hide pointers in code behind a layer of indirection [3]. In another recent work, Crane et. al developed a framework called *Readactor* which aims to be resilient against memory disclosures and aims to provide a high degree of protection against code-reuse attacks of all kinds [13]. Code pointers in code are not readable, as code is mapped as execute-only, and code-pointers in data are replaced by execute-only trampolines to their appropriate functions. However, the authors note that hidden functions which are imported from other modules can be invoked by an adversary through the trampolines if she manages to disclose trampoline addresses. Based on Readactor, *Readactor++* was developed which additionally randomizes the entries in function tables such as in *virtual function tables* and *procedure linkage tables* [14]. Export symbols, however, are and must remain discoverable (see Section III-F on dynamic loading for details).

We show in Section V that this leaves enough space to conduct powerful code-reuse attacks, when combined with crash-resistance. Additionally, we found that it is challenging to hide pointers in structures which are allowed to be accessed legitimately (see Section III-F for details).

III. Unveiling Hidden Memory

In the following, we demonstrate that a memory scanning ability can be achieved by abusing the fact that certain code constructs enable a crash-resistance. We introduce the technical building blocks and show how they can be used to subvert memory secrecy and randomization *without* control-flow hijacking, code-injection, or code-reuse attacks.

A. Fault-Tolerant Functionality

Querying characteristics of memory regions is a legitimate operation in a standard user-mode program. For example, Windows provides API functions for that purpose: `IsBadReadPtr()` and related functions allow a programmer to investigate if a certain memory pointer is accessible with certain permissions without raising faults. Similarly, `VirtualQuery()` yields memory information of a range of pages. Furthermore, other functionality exists whose primary purpose is *not* to deliver information about memory permissions. However, exception handling and system calls can be (ab)used to deduce whether memory is accessible or not.

1) Exception and Signal Handling: Program code in Windows can be guarded via *Structured Exception Handling* (SEH) [41, 45] and *Vectored Exception Handling* (VEH) [42]. A programmer can install *exception handlers* and define *filter functions* which decide if the handler is executed. In case of C/C++ code and SEH, this is achieved with `__try{...} __except(FILTER){...}` and similar constructs, and in case of VEH with the Windows API. This way, a chain of exception handlers can be constructed: If an exception like an access violation is raised, the exception handlers' filters are inspected successively until one handler is picked to process the exception. It can then decide to pass the exception to the next exception handler, terminate the program, or return a status, such that program execution is resumed. In case of SEH, program resumption can continue to execute the code which follows the `__except(){}` block, and in case of VEH, the program is resumed at an address specified within the VEH information. Signal handling is achieved in a similar way in Linux: callback functions can be specified which are called upon a signal raised by the program, such as a segmentation fault. Similar to Windows, the callback function can process the reason for the signal and decide to terminate the program or to resume normal execution.

As we demonstrate in the following, legitimate exception handling can be utilized to achieve crash-resistant functionality within a higher-level interpreter language like JavaScript in a browser *without* hijacking the control-flow.

2) System Calls: System calls in Linux have the ability to return specific status codes based on the parameters they were called with. If a system call expects a pointer to memory and receives a pointer to an unreadable memory address, it will return a different status code than when called with a parameter which points to a readable memory address. For example, the `access()` system call in Linux is normally used to check different characteristics of a file whose name is passed as a string pointer. If the pointer points to an unreadable memory

```
1  #include<windows.h>
2  #include<stdio.h>
3
4  PCHAR ptr = 0;
5  typedef VOID (*function)();
6
7  VOID CALLBACK triggerFault(){
8      CHAR mem;
9      ptr++;
10     switch ((INT)ptr % 3){
11     case 0:
12         printf("Execute 0x%.8x\n", ptr);
13         ((function)(ptr))();
14         break;
15     case 1:
16         printf("Read at 0x%.8x\n", ptr);
17         mem = *ptr;
18         break;
19     case 2:
20         printf("Write to 0x%.8x\n", ptr);
21         *ptr = 0;
22         break;
23     }
24     printf("No fault");
25 }
26
27 INT main(){
28     MSG msg;
29     SetTimer(0, 0, 1000, (TIMERPROC)triggerFault);
30     while(1){
31         GetMessage(&msg, NULL, 0, 0);
32         DispatchMessage(&msg);
33     }
34 }
```

Listing 1. Crash-resistant program in Windows

page, `access()` returns the error "*Bad Address*", while it returns "*No such file or directory*" for a readable memory address (which does not have to constitute a valid filename). A similar behaviour can be observed with system calls on Windows. The system call `NtReadVirtualMemory()` returns a different status code when applied to a readable memory address than to an unreadable memory address.

However, both Windows and Linux do not raise any exception or access fault. This side channel is still used by *egghunt* shellcode. It is a specific type of injected code which searches the memory space for the actual malware code *after* the control-flow was hijacked with a vulnerability [43]. In this paper, we show that actually searching the memory space is possible *without* control-flow hijacking.

B. Crash-Resistance

Memory access faults like memory access violations in Windows programs or segmentation faults in Linux programs are fatal and lead to the abnormal termination of the program. In both operating systems, exception handling is allowed to inspect the type, reason, and faulting code which caused the exception. If the faulting code is not handled by any exception handler, the OS terminates the program. Surprisingly, we discovered that faulting code which should crash a given program does not have to bring down the program necessarily. If we can force a program to stay alive *despite* its code producing memory corruptions and access faults, we denote this as *crash-resistance*.

Consider for example the Windows C-code in Listing 1. On line 29, the timer callback function `triggerFault()` is installed. It is executed in a loop with `DispatchMessage()` (line 30 to 33). `triggerFault()` generates read, write, and execution faults depending on the value of `ptr` which is increased each time it runs. There are no custom SEH or VEH handlers installed, thus the OS should terminate the program on the first access fault. However, this is not the case: the function `triggerFault()` is stopped at access faults, but is executed permanently anew. Hence, `ptr` is continuously increased and each access fault is triggered *without* forcing the program into termination. Consequently, the program is *crash-resistant*.

This behaviour was observed for both 32-bit and 64-bit programs and we found the following reasons for it: the timer callback `triggerFault()` is called by the function `DispatchMessagerWorker()` from `user32.dll`. The callback is wrapped by an exception handler. If an exception in `triggerFault()` is raised, the corresponding filter function executes and decides if the installed exception handler is going to handle the exception. The filter returns `EXCEPTION_EXECUTE_HANDLER` *independently* of the exception type. This instructs the exception handler to handle *any* exception. `DispatchMessagerWorker()` returns and the program continues running without executing line 24.

After coorperation with Microsoft, this issue was confirmed to be security relevant (tracked as CVE-2015-6161, see Section VII-B for a discussion). There exist similar design choices and a more in depth technical analysis can be found in an article by Permamedov [40].

1) Crash-Resistance in Microsoft Internet Explorer: It is important to note that we can exploit this feature inside Internet Explorer and prevent it from abnormal termination on memory corruption errors. We developed two ways to achieve this behaviour in Internet Explorer:

1) A web page can use the JavaScript method `window.open()` to open a new browser tab window. JavaScript code which is dispatched via `setTimeout()` or `setInterval()` inside that window can produce memory corruptions without forcing the browser to terminate.
2) Since the introduction of HTML5, *web workers* are available and dispatched as real threads in script engines. JavaScript code executed with `setTimeout()` or `setInterval()` inside web workers can generate access faults without crashing Internet Explorer.

2) Crash-Resistance in Mozilla Firefox: While the crash-resistance in Windows may seem like an obscure feature, we were able to achieve crash-resistance in Mozilla Firefox as well. We utilized the Firefox JavaScript engine *SpiderMonkey* and its *asm.js* optimization module, called *OdinMonkey*. It is able to compile a subset of JavaScript code ahead of time into high performance native code [33].

We observed that *OdinMonkey* uses exceptions instead of runtime checks in special cases. Most prominently, bounds checking is not performed explicitly. Instead, page protections on memory are used to check bounds implicitly. Every *asm.js* function can access a pre-determined heap of a fixed size. On creation, the heap is initialized as an array with zeros. Thus, any access in bounds will not lead to an exception. On 64-bit machines it is then guarded by a non-accessible memory region of slightly more than 4GB. As *asm.js* only

permits 32-bit indices, this guarantees that any offset from the beginning of the heap will either point into the valid array or into the guard region. Out of bound memory accesses on that array are not treated as critical faults. Instead, a *default* value of `NaN` is returned to indicate that an element outside of that array was accessed. This is accomplished by an exception handler which prevents program termination: OdinMonkey sets a global signal handler that gets called for every unhandled exception in the process. The handler is defined in `AsmJSSignalHandlers.cpp`. Out of bound memory accesses provoke checks to ensure only the intended faults are caught. First, the exception code itself is inspected to determine if it is really an access violation. The faulting instruction address is checked against the location of the asm.js compiled code to ensure that it has thrown the exception. The last check determines if the accessed address lies within the heap and the guard pages of the asm.js code, but outside of the bounds indicated by the size of the array buffer. Only if these conditions are met, the handler signals that the exception has been successfully resolved. It sets the instruction pointer to the instruction following the one causing the fault and sets the default value to be returned. Execution can continue safely as if the access occurred correctly. The asm.js generated code can then perform calculations with the default value or return it into the fully featured JavaScript context. Setting the asm.js heap pointer with a vulnerability is sufficient to achieve crash-resistance in Firefox. Accesses to unmapped memory are then treated as standard out of bound array accesses.

C. Memory Oracles

Armed with crash-resistance, we are able to develop a novel memory probing method for web browsers. We denote *memory oracles* to accomplish the following functionality within JavaScript:

- If non-readable memory is accessed with read access, an access fault is generated and handled in a way that allows recognizing this event.
- In case memory is successfully read, the oracle returns the bytes at that memory location.

In the following, we present the basic design of our memory oracles. Due to the differences between the two scripting engines within Internet Explorer and Mozilla Firefox, the technical implementations differ, but the general approach and the end result are the same for both browsers.

1) Memory Oracles for Internet Explorer: Assume an adversary controls the buffer pointer in a string object by a vulnerability. She can misuse that string object as a memory oracle as shown in the HTML Listing 2: In line 7 of `oracle.html`, a JavaScript string pointing to the four-byte sized wide char buffer `"AB"` is allocated. Then, it is modified with a vulnerability by the attacker on line 8 to a memory address whose permissions are uncertain. This is only illustrated with a comment in Listing 2. Thus, `strObj` does not point to the actual data (`"AB"`) anymore, but to an attacker-chosen address. Line 5 of `runOracle.html` dispatches the JavaScript function `memoryOracle()` in crash-resistant mode. If the modified pointer points to unreadable memory, then `memoryOracle()` stops running, but Internet Explorer stays alive. Thus, the oracle can be queried again with another

```
1  <!-- file oracle.html -->
2  <script>
3  function memoryOracle(){
4      mem = strObj.substring(0,1);
5      /* continue computation */
6  }
7  var strObj = "AB"
8  /* modify WCHAR buffer pointer of strObj */
9  var mem = undefined;
10 window.open("runOracle.html");
11 </script>
```

```
1  <!-- file runOracle.html -->
2  <body onload=runOracle()>
3  <script>
4  function runOracle(){
5      setTimeout("window.opener.memoryOracle();", 0)
6  }
7  </script>
8  </body>
```

Listing 2. Memory oracle in Internet Explorer 11. `oracle.html` is used to open `runOracle.html`

pointer value. If a readable address is found, two bytes are returned and further computations are carried out (line 5 in `oracle.html`). Note that memory oracles can be seen as an extension of memory disclosures, but are more powerful as they can discover reference-less memory.

2) Memory Oracles for Mozilla Firefox: As mentioned earlier we use asm.js to implement our memory oracle for Mozilla Firefox. Due to the extensive checks performed by this browser, developing a memory oracle is more complex than in Internet Explorer. An object of type `AsmJSModule` tracks all information related to an asm.js-compiled module. This includes the location of the native code as well as the asm.js heap location. As mentioned earlier, we do not only need to perform our invalid access from an asm.js function, but also are limited to the heap location plus the size of the guard region. But with a vulnerability, the location of the `AsmJSModule` object is disclosed, as it is reachable with memory disclosures (see Section V-B). Then, the heap address stored in the object's metadata is overwritten to an attacker-chosen address. A read attempt via an array access yields either the default value or content at that address. The former is only retrieved if memory is not readable. Hence, this constitutes already our basic memory oracle. To query the oracle again, the heap address in the `AsmJSModule` object is set to another value and an array access is performed anew. As we will demonstrate in Section V-B, the complete virtual address space can be probed continuously.

D. Web Workers as Probing Agents

Web Workers are a feature of modern browsers. They are intended to run as separate threads in a script environment. We found that web workers can also be used as memory oracles since they can be made crash-resistant. We developed a way to utilize web workers to deduce information whether memory is accessible or not.

In Listing 3, an attacker can control the wide char buffer pointer of object `strObj` with the first element of the array object `bufPtr`. Triggering the vulnerability and initializing `bufPtr` is omitted in Listing 3, but as we show in Section V, such a powerful control is realistic and can be achieved with

```
<!-- file main.html -->
<script>
w = new Worker("worker.js")
// register worker message receiver
w.onmessage = function(e){
    handleMessageFromWorker(e)
}
// start web worker
w.postMessage("startWorker")
function handleMessageFromWorker(e){
    /* continue computation */
}
</script>
```

```
// file worker.js
self.addEventListener('message', initProbe, true)
function initProbe(){
    strObj = "AB"
    pageStep = 0x1000; pageCount = 0
    idProbeMemory = setInterval(probeMemory, 0)
}
function probeMemory(){
    addr = pageStep * pageCount
    /* increase WCHAR ptr of strObj via bufPtr */
    bufPtr[0] = addr
    pageCount++
    /* try to read at address bufPtr[0] */
    mem = strObj.substring(0,2)
    /* return here only if addr was readable */
    clearInterval(idProbeMemory)
    postMessage({ firstPage: addr, content: mem })
}
```

Listing 3. Using web workers to find the first readable memory page in Internet Explorer 11

a single memory corruption such as a null byte write or a use-after-free vulnerability.

The web worker is started on line 9 in `main.html`. On line 6 of worker.js, the function `probeMemory()` is dispatched in crash-resistant mode with `setInterval()`. This causes `probeMemory()` to start subsequently anew, but it stops at line 14 due to read access faults. It only runs further if the read attempt on line 14 succeeds. This occurs eventually: as the read attempt starts at address `0x00` but is increased by `0x1000` bytes on each run, four bytes of the first memory page are returned finally. The content is transfered from the worker to the context of `main.html` on line 17 and can be processed further in `handleMessageFromWorker()` in `main.html`.

E. Finding Unreachable Memory Regions

With the ability to probe memory in browsers, we can discover hidden memory areas like the Thread Environment Block (TEB) or the safe region used by CPI to store pointer metadata. Note that no references to these structures exist in memory, and hence, they are not locatable by simple memory disclosure attacks. The intuition behind our attack is that we can probe for specific information and these probes enable us to deduce if we have found the correct region. We thereby neither use control flow-hijacking nor code-reuse nor code-injection techniques as part of our attack.

We first explain how we can find the TEB. In 32-bit processes, the structure within a TEB from offset `0x00` to `0x18` is known as *Thread Information Block* (TIB) and contains a pointer to `ExceptionList` at offset `0x00`. This pointer points into a thread's stack, because the OS places at least one exception structure on the stack. Thus, the pointer's value is

Algorithm 1: Discover a TEB via memory oracles

Data: Globals: $addrToProbe, pageCount, pageStep, tebMaxEnd, idGetTEB, teb$
Result: address of TEB in teb
Function *startProbe*
  $pageStep \leftarrow 0x1000$
  $tebMaxEnd \leftarrow 0x80000000 - 4$
  $pageCount \leftarrow 0$
  $idGetTEB \leftarrow setInterval(getTEB, 0)$
end
Function *getTEB*
  $addrToProbe \leftarrow (tebMaxEnd - pageStep \times pageCount)$
  $pageCount \leftarrow pageCount + 1$
  oracleProbe $addrToProbe$

  /* at this point probing succeeded */
  $clearInterval(idGetTEB)$
  $teb \leftarrow$ **setToPageBegin** $addrToProbe$

  /* read TEB specific fields */
  $ExcList \leftarrow readDword(teb)$
  $StackBase \leftarrow readDword(teb + 4)$
  $StackLimit \leftarrow readDword(teb + 8)$
  $tebSelf \leftarrow readDword(teb + 0x18)$

  /* heuristic to identify TEB */
  bool $isTEB \leftarrow (teb == tebSelf)$
  if $isTEB \wedge (ExcList < StackBase) \wedge (ExcList > StackLimit)$
  then
    $success = 1$
  else
    /* we found other readable memory */
    /* continue probing for a TEB */
    $idGetTEB \leftarrow setInterval(getTEB, 0)$
  end
end

between the values `StackBase` and `StackLimit` at offset `0x04` and `0x08`, respectively. Additionally, the field at `0x18` contains the address of the TEB/TIB itself.

Thus, we can apply a simple heuristic to scan over the memory space and discover a TEB (see Algorithm 1). Probing for a TEB in a 32-bit process (e.g., Internet Explorer tab process) starts at the end of the last usermode page `0x7ffffffc` ($tebMaxEnd$). TEBs can reside somewhere in the address space between `0x78000000` to `0x80000000` [45]. No other structures except for the PEB and shared data are in that memory region. The call to $setInterval()$ in $startProbe()$ sets $getTEB()$ as timed function to execute permanently anew. An address is queried with a memory oracle ($oracleProbe()$) which either returns when the address is readable, or produces an access fault. In the latter case, $getTEB()$ executes again and the address to probe is decreased by the size of a memory page. As soon as an address is readable, its first three least significant bytes are set to zero ($setToPageBegin()$). The timed execution of $getTEB()$ is cleared with $clearInterval()$ and specific fields are read via memory disclosures. If the fields conform to a TEB structure, then $success$ is set, otherwise $setInterval()$ sets $getTEB()$ again to be executed in intervals. On success, the adversary can read any TEB or PEB information to abuse them in malicious computations further on.

The same method can be applied to 64-bit processes as well to discover the TEB: the offsets have to be adjusted to conform with the 64-bit pointer size and the address of the last possible usermode page, where probing starts, has to be modified. The algorithm can be extended to probe fields of a PEB in case the TEB heuristic triggers. This avoids false positives, which may be hit on 64-bit, as TEBs are mapped below shared libraries.

1) Discovering CPI Safe Region: The linear table-based and hashtable-based 64-bit implementations of CPI rely on hiding the location of the safe region from an attacker [27]. In the linear table-based implementation, the safe region is 2^{42} Bytes (4 TiB) in size, out of the 2^{47} Bytes (128 TiB) of available virtual userspace memory on modern x86-64 processors. Trivially an attacker can guess any address inside the safe region with a probability of 3.125%, but has no way of knowing where exactly this address is located in relation to the start of the region. Thus, she cannot deduce where the metadata for a specific pointer resides. Without a memory oracle, this provides an acceptable level of security. However, an attacker capable of probing memory can quickly find the exact location of the safe region without the risk of crashing the process.

The safe region consists of mostly zero bytes pagewise. Thus, we can distinguish a non-mapped address from an address containing one or more zero bytes. We use an approach that merely scans for zero bytes. If it locates a mapped address, it samples more addresses in the same page. This determines whether it is part of the safe region or if a false positive was hit. Due to the sparsely populated region, this yields correct results under nearly all circumstances. Evans et al. [19] also observed this behaviour in their work.

After we hit the safe region, we still have no knowledge about where it exactly begins. As we can safely cause access violations due to the crash-resistance, we employ a binary search downward from this address until we find the first page. The algorithm works due to the fact that access to an address before the safe region will cause a fault, while an access anywhere after the start of the mapped area will not cause a fault. Consequently, we can approximate the beginning of the region by halving the error margin with every step. The maximum number of probes with binary search is $\log_2 n$, with n being the number of elements to search in. There are $4TiB/4KiB = 1,073,741,824$ possible pages containing the start of the safe region. This means we need a maximum of $\log_2 1,073,741,824 = 30$ tries after we located an arbitrary address in the safe region to determine the start. Assuming the worst case, we need 32 ($128TiB/4TiB$) probes to locate the safe region and afterwards 30 probes to locate the exact starting address. To decrease the likelihood of erroneously marking an address containing zero bytes not belonging to the safe region, the algorithm can be modified to sample more addresses in the same page.

With the ability to alter the information on any pointer we want, the protection of CPI can be circumvented as we can just set the value allowing the action we need to perform with the pointer. Note that the attack assumptions (i.e., a read and write primitive as well as an information leak) required for our memory oracle are within the threat model of CPI.

F. Subverting Hidden Code Layouts

Data structures related to exports are an essential aspect of dynamic loading. These data structures contain function addresses allowed to be imported by other modules, as explained in the following. We first cover this background information before discussing which challenges this introduces for defenses.

1) Dynamic Loading: Windows as well as Linux provide legitimate methods to load shared libraries into a *running* process. This procedure is known as *dynamic loading*. Shared libraries in Windows contain an *Export Address Table* (EAT) with pointers of exportable functions. This structure is often accessed by legitimate code even during a program's runtime and not only at load time. For example, the Windows API function `GetProcAddress()` solely needs the module base and the function name to retrieve a function address. It reads the module's *Portable Executable* (PE) metadata until it discovers the appropriate function and returns its address. Hence, knowing a module's base address is sufficient to retrieve any of its exportable functions. Linux provides a similar API: `dlopen()` can be used to load a shared library into a running process and `dlsym()` returns the address of a needed symbol (e.g., a function).

The key observation is that export symbols and export addresses are available throughout the complete runtime of a process. This is necessary because a library loaded dynamically during runtime may *import* functions which are exported by system libraries like `ntdll.dll` (Windows) or `libc.so` (Linux). Therefore, exports in system libraries are inevitable. Dynamic loading is especially important in web browsers: Firefox implements a plugin architecture to load desired features on the fly. Similarly, Windows implements the *Component Object Model* (COM) which is indispensable for *ActiveX* plugins in Internet Explorer [11].

Note that disabling dynamic loading is not an option in practice, as it would break fundamental functionality and compatibility, and would require loading all libraries at startup of a process. To the best of our knowledge, there is no defense which protects export symbols against illegal access. However, *Export Address Table filtering Plus* (EAF+) of EMET [31] forbids reading export structures based on the origin of the read instruction. We show in Section V-A that this is only a small hurdle in practice.

2) Leveraging Crash-Resistance to Subvert Hidden Code Layouts: In case of code pointer hiding, which is utilized by *Readactor* [13], the functions' addresses are hidden behind execute-only trampolines which mediate execution to the appropriate functions. Thus, their start addresses cannot be read directly. However, with crash-resistance, an adversary can discover the TEB without control-flow hijacking. After she reads information of a TEB, she can read the base addresses of all modules out of the PEB. Another option despite TEB discovery is to sweep through the address space in crash-resistant mode. As the PE file header starting at a module's base is characteristic, memory oracles can provide modules' base addresses. Furthermore, by utilizing memory disclosures, the attacker can resolve trampoline addresses corresponding to exported functions. She can then chain together several trampoline addresses to perform whole function code-reuse, as we will demonstrate later on.

IV. Conquering (Re-)Randomization

Randomization of the memory layout or the code itself has been proposed by various works (e.g., [23, 37, 56]) and much attention was payed to their security and effectiveness. Thus, the latest outcome of this evolution are fine-grained re-randomization schemes such as *Isomeron* [17], which aims

at preventing code-reuse primitives (see Section II-B3 for details). When utilizing crash-resistance, an adversary can abuse weak points in the defenses.

A. Defeating Fine-Grained Re-Randomization

In the case of *Isomeron*, re-randomization is applied to the layout of the code. Hence, in two different points of time one of two different code versions can be used for a specific execution flow. However, to the best of our knowledge, data is not re-randomized at all. Thus, constant data is a foothold for an adversary to undermine the security guarantees of Isomeron as we discuss in the following.

1) Constant Structures: As explained above, the knowledge about a module's base address is sufficient to resolve any of its exported functions. While re-randomizing the code layout during runtime can be performed efficiently, re-randomizing the layout of data structures and the base addresses of modules has yet to be shown. Moreover, the PE metadata layout of a module needed to discover export functions *must* stay consistent such that legitimate code can traverse it. Dynamic loading crucially relies on this aspect. The same holds for the metadata of the *ELF* file format. The potential shortcoming is that an attacker can read that metadata with memory disclosures as well. Re-randomizing the metadata such that its field offsets change would require adjusting legitimate code which accesses it. Additionally, data structures in other modules which reference the metadata need to be updated, too. Thus, we assume re-randomization of data structures allocated in a large number across the complete virtual address space is a challenging and yet unsolved task.

2) Pulling Sensitive Information: A TEB also contains a pointer to a process' PEB. One of its fields, namely `PPEB_LDR_DATA LoaderData`, contains the base addresses, names, and entry points of all loaded modules. We extract that information after we found a TEB with memory oracles. Then we can traverse the PE metadata of each module and retrieve all exported functions *independent* of the randomization applied. We therefore read the individual PE fields with memory disclosures and follow the specific offsets until we reach the EAT. Then, we can loop over the function names and extract the function addresses. As noted above, EMET applies filters to EATs, such that only legitimate code can traverse it. These are ineffective in practice and can be bypassed as we show in Section V-A.

B. Code Execution under Re-Randomizing and CFI

Abusing a memory corruption vulnerability in an address space which is randomized in a fine-grained way on the instruction level and additionally re-randomizes before each function call is very challenging. To further harden exploitation, indirect calls are only allowed to dispatch functions and returns can only target instructions which are preceded by call instructions. This is consistent with coarse-grained CFI like Microsoft's *Control Flow Guard*, BinCFI [60], CCFIR [59] or code-reuse protections in EMET [20, 36]. Thus, known code-reuse primitives such as ROP or Call-Oriented Programming (COP [10]) are not an option. *Return-to-libc* is inappropriate as well since a shadow stack can detect such attacks.

However, as we can retrieve all export functions of all modules via crash-resistance and memory oracles, we opt to chain exported functions in a call-oriented manner. As re-randomization preserves the semantic of functions independent of the code (layout) mutations, they are reusable in a consistent way. The basic idea is to invoke exported functions which dispatch other exported functions on indirect call sites. Ultimately, an adversary can achieve the goal of executing her code of choice.

1) Discovering Functions for Code-Reuse: At the point of control-flow hijacking, when the adversary dispatches her first function of choice, we assume that she can control the first argument's memory (see Section V for details). Thus, we want functions which contain indirect calls whose targets can be controlled with values derived from the first argument. To find possible candidate functions usable for function chaining, we apply static program analysis and symbolic execution.

An executable module is disassembled, its *Control Flow Graph* (CFG) is derived, and all exported functions are discovered. We then mark all indirect calls in them. In the next step, we extract the shortest execution paths between the beginning of a function and its indirect calls. We utilize the symbolic execution functionality of *miasm2* [28] on the gathered paths to detect if the first argument to the function influences the target of the indirect call. If this is the case, we symbolically propagate potential arguments the functions receives to potential parameters a function may take when dispatched at the indirect call site.

Figure 1 illustrates the concept of argument propagation to an indirect call instruction inside `RtlInsertElementGeneric-TableFullAvl` in the NT Layer DLL. ARG_n are arguments the function receives via the stack. At the indirect call site memory at $ARG_1 + 0x2C$ is taken as a call target EIP_{out}. Additionally, arguments are propagated to parameters for the callee (arg_n). For example, the first argument ARG_1 becomes the first parameter for the callee, ARG_3 is increased by ten to become the callee's second parameter arg_2. Such *propagation summaries* for export functions serve as a base to build code-reuse function chains. The ultimate goal is to control the parameters of the *last* function, which eventually performs the operation wanted by an adversary.

2) Crash-Resistant Oriented Programming (CROP): Besides function chaining, an adversary can also utilize the crash-

$$
\begin{aligned}
arg_5 &\leftarrow EBP_{in} \\
arg_4 &\leftarrow EBX_{in} \\
arg_3 &\leftarrow ESI_{in} \\
arg_2 &\leftarrow ARG_3 + 0x10 \\
arg_1 &\leftarrow ARG_1 \\
EAX_{out} &\leftarrow ARG_3 + 0x10 \\
ECX_{out} &\leftarrow ARG_3 \\
EBX_{out} &\leftarrow ARG_1 \\
EIP_{out} &\leftarrow [ARG_1 + 0x2C]
\end{aligned}
$$

Figure 1. Propagation summary for RtlInsertElementGenericTableFullAvl. REG_{in} are registers which are not redefined until the indirect call.

resistance feature to sequentially execute exported system calls or exported functions. Each call is thereby triggered within JavaScript and ends with a fault. As faults are handled, a new call to another exported function or function chain can be prepared and issued as we explain in the following.

The exported function `NtContinue()` in `ntdll.dll` can be used to set a register context [58]. This context is taken as first parameter by `NtContinue()` and registers are set such that program execution continues within that context. At the point of control-flow hijacking which starts a function chain of choice, eventually `NtContinue` is dispatched as the last function in our chain. It takes a propagated argument field as its only parameter `PCONTEXT`. In the `PCONTEXT` parameter, we let the stack pointer point to attacker-controlled memory and the instruction pointer to an exported function like `NtAllocateVirtualMemory()`. The return address for the function is set to `NULL` in the controlled memory. `NtContinue()` sets the register context and the function of choice (e.g., `NtAllocateVirtualMemory()`) executes successfully. Upon its return, an access fault is triggered as it returns to `NULL`. However, this fault is handled and the browser continues running.

This way, exported functions or syscalls can be dispatched subsequently in crash-resistant mode. Similar to our scanning technique shown in Section III-E, this happens within JavaScript with `setTimeout` or `setInterval`. We term this technique *Crash-Resistant Oriented Programming* (CROP) and in spirit it is similar to *sigreturn-oriented programming* (SROP) [8], as we discuss in Section VI.

V. Proof-of-Concept Implementations

To demonstrate the practical viability of the methods discussed in the previous sections, we developed proof-of-concept exploits for Internet Explorer (IE) 10 on Windows 8.0 64-bit, for IE 11 and Firefox 64-bit on Windows 8.1 64-bit, and for Firefox 64-bit on Ubuntu 14.10 Linux 3.17.1. IE is a multi-process architecture whose tab processes run in 32-bit mode. We utilized CVE-2014-0322 which is a use-after-free bug in IE 10. It allows increasing a byte at an attacker controlled address. For IE 11, we utilize an introduced vulnerability which only allows writing a null byte to an attacker-specified address.

The general procedure we utilize to ultimately execute code consists of the following six steps:

1) Trigger the vulnerability to create a read-write primitive usable from JavaScript.
2) Utilize the primitive as memory disclosure feature to leak information *accessible* with memory disclosures.
3) Use the primitive as memory oracles to find constant hidden memory such as the TEB or module bases.
4) Traverse the modules' EATs and extract exported functions.
5) Prepare attacker-controlled objects and set up the function chain.
6) Invoke a JavaScript function to trigger execution of the first function in the chain at an indirect callsite.

A. Exploiting IE without Knowledge of the Memory Layout

We use *heap feng shui* to align objects to predictable addresses [57]. The use-after-free vulnerability in IE 10 and the null byte write in IE 11 are used to modify a JavaScript number inside a JavaScript array. In IE, a generic array keeps array elements in different forms, which depend on their type. Numbers lower than `0x80000000` are stored as $element = number \ll 1 \mid 1$. In contrast, objects are stored as pointers and their least significant bit is never set. We use the vulnerability to modify an element which represents a number. This way, we create a type confusion and let the number point to memory of choice (see Figure 2). We control `0x400` bytes at that location and can read and write it with byte granularity. We craft a fake `Js::LiteralString` object, including the buffer pointer to any address we want, a length field, and the type flag[1]. When the modified number element is accessed, IE will interpret it as a string object. This way, we can use the JavaScript function `escape()` on that element to retrieve the data where the string's pointer is pointing to. This functionality is used to (i) probe addresses we set in our fake string object with crash-resistant memory oracles (see Section III-C) and (ii) read memory content at addresses which are readable.

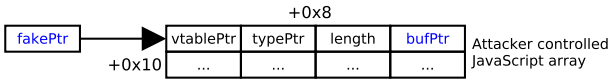

Figure 2. After modifying a number, IE interprets it as pointer to an object (`fakePtr`). As it points to a JavaScript array, elements can be set and fake objects can be created. By varying the buffer pointer (`bufPtr`), a fake string object can be used for crash-resistent memory probing attempts.

1) Memory Probing: After setting the scene, we probe with page-granularity for a TEB starting from `0x7fffffc` and extract addresses of all exported functions. Optionally, we probe with a granularity of 64KB (module alignment) and check for the *DOS header* and *PE header* in case probing returns readable memory. Similar to the former, this circumvents re-randomization schemes which do not re-randomize metadata in a mapped module.

As another hurdle, *Export Address Table Filtering Plus* (EAF+) of Microsoft EMET [31] needs to be bypassed, too, since it checks read attempts to export-metadata of mapped PE modules. If the read originates from illegitimate instructions, then the program is terminated. This should prevent reading export or import metadata with JavaScript. Therefore, modules are blacklisted which are not allowed to access it. However, we discovered that applying `escape()` on large-sized string objects triggers memory copying instructions from whitelisted modules (e.g., `msvcrt.dll`). Thus, we can simply copy a complete PE module into a JavaScript string by using `escape()` on our fake `Js::LiteralString` object. Finally, we can resolve exports within the copy of the module.

2) Code Execution and Function Chaining: With all exported function addresses available, we craft a fake vtable and insert the addresses of five exported functions into the fake object. We dispatch a JavaScript method of the fake string object, which triggers a lookup in the vtable and a dispatch of the first exported function in our chain. Thereby, the first function also receives our fake object as first argument, such that we control two parameters for the last function in the

[1] Most JavaScript objects are C++ objects and contain a *vtable* pointer. As we do not know the location of any module's data yet, we do not set it. However, accessing the fake string object with e.g. `escape()` still works.

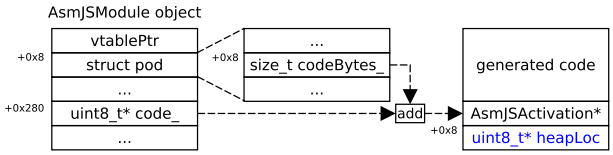

Figure 3. Location of the asm.js heap pointer (`heapLoc`) needed to modify in order to gain crash-resistance.

chain (`LdrInitShimEngineDynamic`). The chain propagates our first controlled argument `P` in a way that

```
LdrInitShimEngineDynamic([P+0x08]+0x20, [P]+0x18)
```

is executed. If the first parameter points to any module, the second parameter can specify a string pointer to a DLL name. This DLL can reside on a remote server and is loaded into the address space of the program. Hence, the adversary reaches her goal of code execution. We opted to use `LdrInitShimEngineDynamic` because neither EMET [31] nor CCFIR [59] blacklist the function, and normally dynamic loading of remote DLLs via the standard Windows API is monitored. We can also use `WinExec([P+0x08]+0x20, <*>)` with a chain of four functions to achieve the execution of arbitrary programs.

B. Memory Probing in Mozilla Firefox

Next, we describe the steps we used to scan memory in 64-bit Mozilla Firefox on Windows 8.1 *and* Ubuntu 14.10 Linux 3.17.1 All steps are also applicable to any other 64-bit application embedding the SpiderMonkey engine. This includes for example Mozilla Thunderbird with asm.js enabled. We introduced a vulnerability into Firefox 38.0 to simulate a real-world bug. This allows to leak information into the JavaScript context and write to memory addresses. With these primitives, we show the creation of crash-resistance and the feasibility of memory oracles to scan arbitrary memory.

In contrast to Internet Explorer, we do not need to rely on `setInterval()` and web workers. Instead of creating a fake object, we need to change fields in Firefox' object metadata to obtain crash-resistance. However, web workers can be used to increase the performance, especially since calling into and out of asm.js is an expensive operation. The main bottleneck is the handling of generated faults, because they are delivered with four context switches for every exception.

a) Manipulating the Function Object: We let Firefox create an asm.js function by utilizing the asm.js subset of JavaScript. This leads to a `AsmJSModule` memory object. We then overwrite the heap location to point to the memory region we want to scan. To achieve this goal, we use an information leak to first deduce the location of the `JSSValue` object, which constitutes the function reference. Then, we utilize targeted reads to learn the location of the heap address in the `AsmJSModule`. We then set the heap location to the region we want to scan (see Figure 3). It is possible to use the function object several times: by setting the heap location to another address for each probe, we use the ahead-of-time compiled asm.js code repeatedly.

b) Probing the Memory: A loop in JavaScript is utilized which calls into asm.js and utilizes the asm.js crash-resistant functionality to probe target regions. We use an asm.js function which returns a 64-bit double float value at a given address. The target region may contain entries that are interpreted as `NaN`. Due to the way floating point numbers are handled, the result is `NaN` if a value with the highest 11 bits set to 1 is hit. This cannot be distinguished from a faulting read attempt, as this yields `NaN` as well. This needs additional byte-shifted probes around that address to verify that the page is indeed mapped. By retrieving a value which is not `NaN`, we are certain that the page is mapped.

We are able to scan 4GiB beginning from the heap start. Once this space is scanned, the heap address of the asm.js module needs to be adjusted and scanning can continue. Care has to be taken to only perform probes which attempt to read out of bounds of the asm.js heap size. The heap size is specified on creation of the asm.js object. Normally, when the heap pointer is not modified, inbound accesses do not throw an exception, while out of bound accesses do. As we modified the heap location to point to an unmapped address, inbound heap accesses will throw exceptions. These are not crash-resistant. Simply moving the initial heap location to a lower address and scanning with an offset from the targeted address avoids this problem. Thus, only out of bound probes are utilized, as these do not terminate Firefox.

c) Determining Memory Contents: Once a mapped page is found, we need to determine what it contains. When sweeping the complete memory, we can hit shared modules, data structures like the TEB, or application heaps. Learning what memory contains can be done using regular JavaScript by utilizing the capability to call fully featured JavaScript functions from asm.js. We can therefore use the same heuristics used with Internet Explorer to, for example, safely deduce a TEB. At the end, the same techniques used in Internet Explorer can be utilized to gain code execution.

C. Memory Scan Timings

We used `performance.now()`, the high-resolution performance counter of JavaScript. We performed 268,369,911 probes with Firefox (Windows and Linux) and 32,768 probes with IE on *unmapped* memory to measure the probes per seconds a single browser thread can achieve. We observed different scanning rates between the two tested browsers. 32-bit Internet Explorer was only able to reach 63 probes/s, while Firefox was able to scan with 718 probes/s in Windows and 18,357 probes/s in Linux on average. However, this includes optimizations as explained below.

The difference in scanning speeds is caused by the different methods used for probing. While Internet Explorer needs to spawn a new JavaScript function with `setInterval()` for every probe, Firefox uses ahead-of-time compiled code of asm.js. This causes significantly less overhead. We were able to move parts of the JavaScript scanning loop into the asm.js code, providing another speedup. This is due to fewer calls into the asm.js JavaScript subset, which are expensive. We provide only maximum scanning times, as these are already small enough to be practical for an adversary.

In 32-bit Internet Explorer it takes at most $(0x80000000 - 0x78000000)/0x1000 = 32,768$ crash-resistant probes to locate the TEB. Thus, the maximum scanning time is $(32,768/63)s = 520.1s$ (8.7 minutes). To locate the most upper mapped DLL, $2^8 = 256$ probes are necessary at most. This is due to the module base entropy of 8 bits, the 64KB alignment of modules, and the address scan range from `0x77000000` to `0x78000000` where *at least* one module resides. This yields a maximum scanning time of $(256/63)s = 4.1s$.

In Firefox 64-bit in Windows we scanned for PE metadata of mapped modules. To locate the DLL mapped on top of the address space, it takes at most $2^{19} = 524,288$ probes due to a module base entropy of 19 bits in 64-bit processes. Thus, the maximum scanning time is $(524,288/718)s = 730.2s$ (12.2 minutes). Scanning starts at the top usermode address of `0x7FFFFE0000` and is performed toward lower addresses in 64KB steps.

In Firefox 64-bit for Linux, we focus on finding reference-less hidden memory. An instantiation of reference-less memory is the linear safe region of the 64-bit implementation of CPI. Locating the safe region of CPI can be done in very few steps. As outlined earlier, we first probe for a location in the region and then use binary search to locate the exact starting address. As this requires less than 1,000 probes, it is almost instant $((1,000/18,357)s = 0.05s)$.

The difference in probes/s between the Windows and Linux version of Firefox is due to the fast signal handling in Linux in comparison to the exception handling in Windows. Speed increases further when spawning several workers that perform the scanning. This is due to multiple cores on modern CPUs, which run the worker threads in parallel.

VI. Related Work

Recent years of research show a continuously rising amount of achievements on both the offensive and defensive side. Back in 2004, Shacham et al. [50] showed the ineffectiveness of ASLR on 32-bit systems due to its susceptibility to brute-force attacks. Their work suggested defense mechanisms like subsequent re-randomization. While their approach targeted servers on 32-bit systems, we show that similar capabilities are possible with web browsers on 32-bit and 64-bit platforms.

Since then, several approaches have been proposed to tackle different levels of randomization problems. *Binary Stirring* [56] randomizes basic blocks at the cost of high performance loss. *Oxymoron* approaches the problem by using information hiding techniques [3]. By rewriting 32-bit Linux binaries and adding an additional protected layer of indirection for control flow transfers, Oxymoron allows fine-grained ASLR without losing code sharing capabilities. However, these defenses are still vulnerable against data-only attacks combined with information leakages. Snow et al. introduced just-in-time code-reuse (JIT-ROP) [51] that can repeatedly utilize an information leak to bypass fine-grained ASLR implementations. The authors suggest frequent re-randomization at runtime as a possible solution. *Isomeron* [17] approaches that problem by applying re-randomization at different levels of granularity. This approach has an immense effect on thwarting code-reuse attacks. However, our work shows that re-randomization of code is not enough, as constant structures can be misued to bypass it. Especially with crash-resistance and memory oracles, constant structures are locatable without control-flow hijacking.

Bittau et al. [6] proposed another interesting flavor of ROP attacks which they called *Blind ROP* (BROP). The authors show how stack buffer overflows can be utilized to bypass ASLR and conduct code-reuse attacks remotely. BROP uses server crashes as a side channel which, in turn, reveals information about the memory layout. By locating and arranging specific gadgets remotely, they trigger a write over a socket that transfers the binary to the attacker to find more gadgets. Our work is different in that it focuses on browsers, which have a hard crash policy. Nevertheless, with crash-resistance and memory oracles we are able to undermine memory secrecy.

Seibert et al. introduced another approach on the Apache server by reading bytes and measuring the time [48]. It turns out that specific bytes leave different timing patterns and thus, probed in sequence, reveal information about the memory layout. Our work differs in that we introduce fault-tolerant functionality in browsers, which has not been shown before. Its result, however, is similar, in that we can deduce memory which is not locatable by simple memory disclosures.

Another branch of defense mechanisms involves CFI. Due to its performance overhead, research has put its focus on the coarse-grained variant of CFI. However, recent research demonstrates that the coarse-grained variants are prone to code-reuse attacks [16, 21, 47]. Schuster et al. introduce *Counterfeit Object-Oriented Programming* (COOP) [46] that ranks itself on the same line with other code-reuse attacks. The authors manage to bypass many CFI defense mechanisms by using chains of existing C++ virtual function calls. The drawback is that semantic-aware C++ defenses check virtual function table hierarchies and prevent COOP. In contrast, we present a different function-reuse technique in addition to the contributions of crash-resistance and memory oracles. It uses exported function chains and C-like indirect calls instead of virtual function calls. Thus, C++ defenses are insufficient against it. Furthermore, we combined function chaining with fault-tolerance to gain a novel function-reuse technique named *Crash-Resistant Oriented Programming* (CROP).

Note that register or data-flow randomization is insufficient as a defense against function chaining as well: function prototypes of exported functions are mostly documented to ease their usage by a programmer. Thus, the number of arguments and their types are known. If an exported function propagates fields of its first argument structure to parameters for a function at an indirect call site, the propagation is unaffected by register or data-flow randomization. As the propagated fields constitute parameters, they always need to be pushed onto the stack or put into parameter registers specified in the ABI. Shuffling the parameters makes is necessary to adjust the parameter handling of each function which is allowed at the indirect call site. To our knowledge this is not done by current defenses [13, 37].

In 2014, Kuznetsov et al. introduce CPI [26]. As discussed before, CPI is prone to data pointer overwrites: Evans et al. showed that such overwrites can be utilized to launch timing side-channel attacks that lead to information leakages about the safe region [19]. Similarily, we can deduce the reference-

less safe region. However, we show that it is possible within Firefox which does normally not allow faults, while Evans et al. utilize the web server *Nginx*, which respawns upon a crash.

CROP is in spirit similar to *sigreturn-oriented programming* (SROP [8]), as we can set register contexts as well. While SROP is only possible on Linux, we can utilize crash-resistance to perform arbitrary exported function chaining and system call dispatching on Windows in a fault-tolerant way.

VII. Discussion

In the following, we discuss the implications and limitations of crash-resistance and memory oracles. Additionally, we elaborate on potential countermeasures and design choices to thwart fault-tolerant memory scanning.

A. Novel Memory Scanning Technique

The existence of reliable and fast memory oracles enables an attacker to bypass all defenses that rely in any way on metadata that is stored in userspace. A common approach was to keep the data in reference-less memory so an attacker would need to hijack the control flow, inject code, or perform code-reuse attacks, before disabling that protection. This implies that the defense also protected itself. However, we show that hidden information in the userspace can be found by an attacker, without control-flow hijacking, code-injection or code-reuse attacks. While this primitive alone does not allow an attacker to exploit an application, it provides a valuable addition to her arsenal. It is an advantage when simple memory disclosures are not an option. Hence, it might allow circumventing previously effective defense mechanisms.

With the knowledge obtained by crash-resistant address space scanning, an attacker can overwrite data considered to be unreachable by adversaries. If such data serves as metadata for protection mechanisms, it can enable the successful execution of other exploit stages. This might enable control-flow hijacking again or might endanger the security provided by shadow stacks [15]: modifying a reference-less shadow stack after it is discovered with memory oracles might allow traditional code-reuse attacks again.

Another notable example is the reference-less safe region used by 64-bit CPI implementations. CPI is able to prevent control-flow hijacking exploits, but altering the safe region's metadata effectively disables it. This allows control-flow hijacking and thus, the realization of traditional code-reuse attacks such as ROP. However, CPI also provides a *Software Fault Isolation* (SFI [55]) and hash table-based implementation of the safe region [27]. While the SFI version is immune against memory oracles, it has an additional performance overhead of about 5%. The hash table-based version is located in userspace and can have a size of $2^{30.4}$ bytes, while the original linear-based safe region has a size of 2^{42}. According to Kuznetsov et al. [27], it requires around 51,000 probes to locate the hash table-based safe region. In Firefox on 64-bit, we achieve a rate of 18,357 probes per seconds. Thus, locating the safe region would still be fast with only 2.78 seconds. As the 32-bit safe region is protected by segmentation, we cannot reach it with memory oracles.

Recent work named *Readactor++* [14] protects C++ virtual function call sites. We do not claim to have bypassed Readactor++. However, we weakened it in the sense that we can leak information about the memory layout with memory oracles. More specifically, we can extract trampoline addresses corresponding to exported functions.

The speed of memory scanning with memory oracles currently varies across browsers and platforms. This is due to a) the way they are implemented and b) the runtime overhead of the exception/signal handlers. Firefox 64-bit on Linux achieves the fastest scanning as ahead-of-time asm.js code is used, which intentionally uses exceptions for bound checks. Additionally, signal handler on Linux are faster than exception handler on Windows. In contrast, the fault-tolerant feature in Internet Explorer is harnessed with code normally used to execute JavaScript timer functions. Thus, much boilerplate code is executed and slows down the scanning ability, in addition to the SEH exception handling overhead. An increase in performance might be gained with typed arrays, as element accesses map to array element accesses on the assembly level. Currently we use a fake string object in Internet Explorer. With asm.js coming to Internet Explorer on Windows 10 [30], it might be possible to increase the speed further.

We currently only make use of fault-tolerant functionality based on exception/signal handling for crash-resistance, memory oracles, and memory scanning. While we show their existence and powerful advantages, crash-resistance might be achieved with system calls or functions intended to query memory information. We hope that future work will reveal more crash-resistant functionality for different purposes such as CROP (see Section IV-B2). Automated approaches utilizing static analysis might simplify that process, such that legitimate crash-resistant code paths become controllable by an attacker without control-flow hijacking [54].

B. Design Choices, Countermeasures, and Defenses

Several choices can be made to prevent crash-resistance. Single *instances* of crash-resistance are fixable. For example, we do not see any legitimate uses in the crash-resistance of Internet Explorer. Actually, after cooperation with Microsoft it was determined that this issue is security relevant and affects Internet Explorer 7 to 11 and the Microsoft Edge Browser (see CVE-2015-6161). It was fixed for Microsoft Edge during the Patch Tuesday cycle in December 2015 and hardening settings for Internet Explorer were made available [32]. In the past, a few vulnerabilities had the ability to survive crashes, and thus, adversaries were able to trigger them several times in order to bypass ASLR or to increase successful exploit chances [25, 52]. Hence, Microsoft's Security Development Lifecycle (SDL) suggests avoiding global exception handlers which can catch all violations [29]. Note that single instances of buffer overflow vulnerabilities can be fixed as well, while the *class* of buffer overflows cannot be easily completely eliminated. Crash-resistance is similar and we argue that constructing a memory oracle is possible on every modern system which has a way for applications to handle faults.

1) Crash Policies: A general countermeasure is to limit the number of faults that can be caused. This means an attacker must find ways to reduce probing attempts and hit the right location in one of her first scans. However, this only provides a probabilistic solution as there is a small chance for the first

probe to succeed. In addition, a hard crash policy can interfere with use cases where legitimate exceptions can occur and are expected. As described in Section III-B2, Firefox leverages exceptions for fast array accesses to avoid bounds checking. An attempted out of bound read is caught with the help of the exception handler and returns a default value (`undefined`). Removing exception support would decrease the performance, because additional bound checks for every array access would have to be performed.

2) Accurately Checking the Exception Information: The most effective countermeasure against crash-resistance is to accurately check the exception information of a triggered fault. The triggered exceptions we used in Internet Explorer for crash-resistance allow *any* fault to be used as a side-channel. Exception handlers should catch only faults which are expected in guarded code. Therefore, the exception type should be inspected carefully as well as the address of the instruction which caused the fault. This information is necessary to verify that only the intended faults are caught. Additionally, it is necessary to make sure that guarded code cannot throw other exceptions. While it might be difficult to always handle all faults accurately, unintended faults should always be forwarded unhandled. This way the operating system can safely terminate the program and prevent crash-resistance. Note that the fault handler of Firefox performs rigorous checks on the data provided by the OS. This includes information about the address of the instruction causing the fault, the error code, and the exception type. Thus, we needed to modify metadata in the process in order trick the checks before triggering a fault.

Exception information is processed differently in Windows and Linux. Linux can differentiate if a fault occurred due to unmapped memory or due to an access with wrong permissions. As such, exception handling in the asm.js functionality of Firefox can utilize this subtlety to prevent crash-resistance. Actually, this is a good example for fixing a single instance of crash-resistance. Surprisingly, the Firefox developers added a check to the asm.js exception handling in Firefox 39. Accesses to unmapped memory are not handled anymore, but only accesses with wrong permissions. As a guard region follows the asm.js heap, bound checks can still be performed with exceptions. As the fix was not flagged as a security issue, the developers unintentionally eliminated a security issue. However, crash-resistance within asm.js remains in the Windows version of Firefox.

3) Using Guard Pages to Prevent Probing: We realized in our tests with Firefox in Windows that accesses to guard pages around the stacks were not crash-resistant. Guard pages around the stack normally prevent stack overflows. An access to a guard page delivered an error code different to the error code of a heap guard region access. This was not handled by the asm.js handler. By placing guard pages around critical structures, scanning attempts performed by an attacker can be detected. The program can then be terminated immediately whenever an illegal access is detected. The difference in the exception code allows distinguishing potentially intended faults from exceptions caused by an attacker. However, the same fault in Internet Explorer still allows complete crash-resistant memory scanning as any fault is handled. Thus, probing an unmapped page yields the same result as touching a guard page.

4) Defenses against Crash-Resistance: Softbound [34] and CETS [35] are memory corruption defenses and memory safety solutions for C programs. The former provides spatial safety, while the latter prevents temporal bugs. As memory corruptions are eliminated, our current approach of crash-resistance is not possible in C programs. However, most parts of Firefox and Internet Explorer are written in C++, which Softbound and CETS do not support.

VIII. Conclusion

In this paper, we demonstrated that even client applications such as web browsers can be resistant to crashes. We showed that an adversary can safely query the address space, which is normally not legitimate and should lead to program termination. We thereby do not rely on control-flow hijacking, code-injection or code-reuse attacks. To this end, we introduced the concept of *crash-resistance* and developed *memory oracles*. This enables an adversary to use fault-tolerant functionality as a side channel to obtain information about the memory layout. Furthermore, we introduced the concept of *Crash-Resistant Oriented Programming* (CROP) that leverages *crash-resistance* to execute function chains in a fault-tolerant manner. As a result, recently proposed information hiding and randomization defenses are weakened, and control-flow hijacking and code-reuse attacks can be enabled again.

Acknowledgment

This material is based upon work partially supported by 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] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

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

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*, 2003.

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

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

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. In *ASIACCS*, 2011.

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

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

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

[11] D. Chappell. *Understanding ActiveX and OLE: a guide for developers and managers*. Microsoft Press, 1996.

[12] 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.

[13] 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.

[14] 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.

[15] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[16] 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.

[17] 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.

[18] 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.

[19] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, 2015.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] K. Kortchinsky. Escaping VMware Workstation through COM1. https://www.exploit-db.com/docs/37276.pdf, 2015.

[26] 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.

[27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity. In *IEEE Symposium on Security and Privacy*, 2015.

[28] miasm2 Authors. Miasm2: Reverse Engineering Framework in Python. https://github.com/cea-sec/miasm, 2015.

[29] Microsoft. The Microsoft SDL and the CWE/SANS Top 25. http://download.microsoft.com/download/C/A/9/CA988ED6-C490-44E9-A8C2-DE098A22080F/Microsoft%20SDL%20and%20the%20CWE-SANS%20Top%2025.doc, 2009.

[30] Microsoft. Bringing asm.js to the Chakra JavaScript engine in Windows 10. http://blogs.msdn.com/b/ie/archive/2015/02/18/bringing-asm-js-to-the-chakra-javascript-engine-in-windows-10.aspx, 2014.

[31] Microsoft. EMET 5.2 is available. http://blogs.technet.com/b/srd/archive/2015/03/16/emet-5-2-is-available.aspx, 2014.

[32] Microsoft. Microsoft Security Bulletin Summary for December 2015. https://technet.microsoft.com/en-us/library/security/ms15-dec.aspx, 2015.

[33] Mozilla. asm.js working draft. http://asmjs.org/spec/latest/.

[34] 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.

[35] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, 2010.

[36] V. Pappas. kBouncer: Efficient and Transparent ROP Mitigation. http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf.

[37] 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.

[38] PaX Team. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt, 2001.

[39] PaX Team. Pageexec. https://pax.grsecurity.net/docs/pageexec.txt, 2001.

[40] A. Permamedov. Why it's not crashing? *The Code Project*, 2010.

[41] M. Pietrek. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal-US Edition*, 12(1):41–66, 1997.

[42] M. Pietrek. New vectored exception handling in Windows XP. *MSDN Magazine*, 16(9):131–142, 2001.

[43] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

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

[45] M. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1*. Microsoft Press, 6th edition, 2012.

[46] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy*, 2015.

[47] 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.

[48] 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.

[49] 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.

[50] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[51] 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.

[52] A. Sotirov. Reverse Engineering and the ANI Vulnerability. http://www.phreedom.org/presentations/reverse-engineering-ani/reverse-engineering-ani.pdf, 2007.

[53] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, 2009.

[54] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic hooks: Hiding control flow changes within non-control data. In *USENIX Security Symposium*, 2014.

[55] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, 1994.

[56] 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.

[57] T. Yan. The Art of Leaks: The Return of Heap Feng Shui. In *CanSecWest*, 2014.

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

[59] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*, 2013.

[60] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.