Shredder: Breaking Exploits through API Specialization

Shachee Mishra
Stony Brook University
shmishra@cs.stonybrook.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

ABSTRACT

Code reuse attacks have been a threat to software security since the introduction of non-executable memory protections. Despite significant advances in various types of additional defenses, such as control flow integrity (CFI) and leakage-resilient code randomization, recent code reuse attacks have demonstrated that these defenses are often not enough to prevent successful exploitation. Sophisticated exploits can reuse code comprising larger code fragments that conform to the enforced CFI policy and which are not affected by randomization.

As a step towards improving our defenses against code reuse attacks, in this paper we present Shredder, a defense-in-depth exploit mitigation tool for the protection of closed-source applications. In a preprocessing phase, Shredder statically analyzes a given application to pinpoint the call sites of potentially useful (to attackers) system API functions, and uses backwards data flow analysis to derive their expected argument values and generate whitelisting policies in a best-effort way. At runtime, using library interposition, Shredder exposes to the protected application only *specialized* versions of these critical API functions, and blocks any invocation that violates the enforced policy. We have experimentally evaluated our prototype implementation for Windows programs using a large set of 251 shellcode and 30 code reuse samples, and show that it improves significantly upon code stripping, a state-of-the-art code surface reduction technique, by blocking a larger number of malicious payloads with negligible runtime overhead.

ACM Reference Format:
Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3274694.3274703

1 INTRODUCTION

After more than two decades of advances in code reuse attacks [24, 36, 51, 52], defending effectively against return-oriented programming (ROP) [63] and its recent more sophisticated forms [12, 15, 21, 26, 59, 61, 65, 66] remains a challenging problem. By diverting the hijacked control flow into carefully selected existing code fragments (dubbed "gadgets") or even whole functions, modern exploits can bypass not only non-executable memory protections, but also (depending on the application and various other conditions) an array of additional exploit mitigation approaches, such as control flow integrity, code diversification, and code pointer protection.

Control flow integrity (CFI) [2] confines control flow transfers within a set of targets, preventing the execution of gadgets that are not part of the legitimate control flow graph. Recent works, however, have shown that exploitation is still possible by transferring control to gadgets or functions without violating the enforced policy [12, 15, 26, 59, 61]. Address space layout randomization (ASLR) [69] randomizes the load address of shared libraries and main executables, while code randomization [44] alters not only the location but also the structure of code, breaking ROP payloads that rely on gadgets present in the original code.

By leveraging a memory leakage vulnerability, however, exploit code can dynamically harvest gadgets and construct a functional "just-in-time" ROP (JIT-ROP) payload for a particular diversified process [65]. Recent leakage-resilient protections against JIT-ROP exploits, which rely on the concept of execute-only memory to block the on-the-fly discovery of gadgets [6, 13, 18, 20, 32, 57, 67, 73], can under certain conditions also be bypassed [59]. Code pointer protections prevent code reuse attacks at their initial step, by preventing the corruption of pointers in memory. Even this strong form of protection, however, can be bypassed when certain code constructs are present, by corrupting non-pointer data [70].

Despite these shortcomings, the continuous deployment of exploit mitigation technologies has undoubtedly been making vulnerability exploitation harder, as evident by the complexity and sophistication of the above exploitation techniques [12, 15, 26, 59, 61, 65, 70]. Especially when orthogonal mitigations are combined (e.g., control flow integrity and code pointer integrity), bypassing all of them becomes challenging even under favorable conditions [70]. Unfortunately, in practice, most of the above protections have not seen widespread deployment due to their reliance on source code or debug symbols, and the often non-negligible runtime overhead they incur. It is thus pertinent to focus part of our defense efforts on approaches that i) can complement existing protection mechanisms to collectively provide stronger protection, ii) can be transparently deployed for the protection of (closed-source) third-party applications, and iii) introduce negligible runtime overhead.

A promising approach that fulfills the above requirements is code surface reduction through the removal of unused code [48, 50, 58]. Code reuse attacks rely on the abundance of code in the address space of a vulnerable process. For most applications, the bulk of this code comes from libraries: DLLs on Windows or shared libraries on UNIX systems. These libraries are either bundled with the application, or are provided by the OS to expose system interfaces and services. Applications typically use only a fraction of these functions, so a natural approach for reducing the code surface of a process is to remove any unneeded functions from its address space after each DLL is loaded [50, 58].

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274703

A particular implementation of this approach for closed-source Windows programs is dubbed *code stripping* [50]. Removing unused system API functions not only reduces the number of gadgets that can potentially be used by an attacker, but also restricts what potentially harmful OS operations (e.g., the allocation of executable memory) can be performed in the first place. For example, server applications often do not need to create *new* network connections (just accept incoming connections); removing `connect()` will thus break any exploit code (either shellcode or ROP code) that attempts to connect back to an attacker-controlled system [50].

As an added benefit, the removal of unused code can eliminate potential vulnerabilities due to bugs in "forgotten" parts of the code. At the same time, this approach does not incur any additional overhead (in fact, it may slightly improve performance due to reduced memory utilization), and is applicable to closed-source third-party applications. This is particularly important in the Windows ecosystem, which is probably the most severely plagued by ROP exploits against vulnerable third-party applications—Mulliner and Neugschwandtner [50] have demonstrated the benefits of code stripping for the protection of commonly targeted closed-source Windows applications such as Adobe Reader.

Unfortunately, although unneeded code removal reduces the number of ROP gadgets at the disposal of attackers—and even whole potentially useful (for attackers) system API functions—the code of several system operations that are essential for *both legitimate and malicious* code is unlikely to be removed. For instance, consider `kernel32.dll`, which is typically loaded by all Windows processes, as it provides a host of useful system operations. Among its exported functions are `VirtualAlloc()` and `VirtualProtect()`, which are used by ROP exploits to either allocate writable and executable (WX) memory, or change the permissions of existing memory to WX (to facilitate the execution of second-stage injected shellcode or malicious DLLs). These functions are also indispensable to most programs for generic memory allocation, and thus cannot be simply removed. We observe though that, with a few exceptions, most programs rarely need to allocate *executable* memory—precisely what malicious code needs.

Based on the observation that part of an API function's capabilities may not ever be needed by the target program, in this work we propose *API specialization*, a defense-in-depth approach that restricts the interface of the loaded instances of critical API functions according to the actual needs of a given program. This is achieved by neutralizing any unneeded functionality which may be essential for malicious code, such as the allocation of executable memory (e.g., by not accepting "X" as an allowable permission).

We have developed a prototype of this approach for the protection of closed-source Windows applications, called *Shredder*. In a preprocessing phase, Shredder statically analyzes a given program to pinpoint the call sites of sensitive system API functions, and uses backwards program data flow analysis to derive their expected argument values in a best-effort way. At runtime, using library interposition, Shredder exposes to the protected application only the specialized versions of these functions, and blocks any invocation that violates the enforced policy. We have experimentally evaluated Shredder using a large set of 251 shellcode and 30 ROP payload samples from real-world and proof-of-concept exploits. When applied for the protection of 10 popular Windows 64-bit applications, Shredder blocked significantly higher number of malicious payloads compared to code stripping [50], with negligible runtime overhead.

In summary, our work makes the following main contributions:

- We propose *API specialization*, a best-effort code slimming technique for close-source applications that limits the functionality of system API functions to the absolutely necessary operations that a given program needs.
- We have implemented a prototype tool, call *Shredder*, that employs API specialization to transparently protect Windows applications by blocking the execution of shellcode or ROP code that violates the enforced policy.
- We have experimentally evaluated Shredder with popular Windows applications and a large set of exploit code, and demonstrate that it breaks 18.3% more shellcode and 298% more ROP code samples compared to code stripping [50], while incurring a negligible runtime overhead.

2 BACKGROUND AND MOTIVATION

The first stage of recent exploits (typically implemented as a ROP payload) usually performs a simple task, such as enabling the execution of a more complex second-stage shellcode, or dropping and invoking a malicious executable. This unavoidably requires some interaction with the OS through the system call interface, e.g., to carry out system operations related to file and network activity, memory allocation, and module loading.

In Unix-like systems, the system call interface is exposed to user space through a single library (e.g., `libc`), which provides a mostly one-to-one mapping between the available system calls and the exported API functions that programs can invoke. Windows, in contrast, does not expose system calls to user-level programs in such an obvious and direct way. Instead, programs interact with the OS through the Windows API [47], which is organized into several DLLs. In turn, these DLLs call functions from the (largely undocumented) Native API [60] (implemented in `ntdll.dll`) to invoke kernel-level services.

In both cases, the overall set of system operations available to user programs through the provided APIs is quite large. Depending on the application, however, it is unlikely that all available system operations will be needed. For instance, a program might never need to create network connections or write files to disk. Still, as long as a program needs to perform even a single system operation, it has to import the corresponding function from the respective system library (e.g., `libc.so` in Linux or `kernel32.dll` in Windows), which means that the whole library will be loaded into the process' address space. Although none of the other API functions will ever be used, they are still available in the address space of the process to be used as part of exploit code. As an example, `kernel32.dll` is almost always imported by Windows applications, and provides a large variety of critical functionality for exploit code (e.g., change memory permissions to bypass non-executable memory protections, connect to remote hosts and download malicious components or exfiltrate information, or create new processes and threads to perform malicious actions).

If an API function will never be used by a given application, why making it available to attackers? This is the idea behind *code*

Table 1: Number of imported functions from Windows API DLLs for various 64-bit programs.

DLL name:	kernel32	advapi	user32	ole32
Exported functions:	1941	902	1152	167
Imported functions by:				
7Zip	94	19	84	12
Google Chrome	201	36	33	8
Microsoft Edge	46	-	-	-
Mozilla Firefox	105	29	8	-
iTunes	336	52	212	29
PhotoViewer	115	16	136	23
Notepad++	173	13	177	2
Powershell	80	13	1	6
VLC	38	2	1	-
Winrar	180	26	152	12

stripping [50], which at load time identifies all the *non-imported* functions from external system libraries and removes them from the address space of the protected process. In addition, code stripping is combined with *image freezing*, which prevents the addition of new code, i.e., new executable memory pages, once the process has been initialized. This is achieved by filtering through API hooking the memory-related functions that may receive an "execute" flag. A function that tries to use the "execute" flag succeeds only if it was part of the original program. Both code stripping and image freezing essentially have no runtime overhead, as both are performed only during startup.

Given that invoking system calls directly is usually not an option in Windows exploits (due to the frequent system call number changes across different Windows versions, and the complexity of the Native API [55, 64]), if the API function for a given system operation is missing, any exploit code that relies on it will break. This would prevent attackers from, for instance, accessing the network, if a given application never has to do it. At the same time, by actually removing the code of each unneeded function from the process' memory, the number of ROP gadgets at the disposable of attackers is reduced, while any potentially exploitable bugs present in those functions are also effectively eliminated.

To give an intuition about the great potential of this approach for attack surface reduction, Table 1 shows the number of exported API functions by four commonly used Windows 10 system DLLs (upper part), and the number of actually imported functions from those DLLs by a set of popular Windows applications (lower part). As evident, all applications use only a fraction of the available functions, which means that code stripping can potentially prevent exploits from performing a wide range of system operations.

Unfortunately, however, although part of the system APIs may be neutralized by code stripping, it is very likely that some system functions that may be critical for malicious code will also be needed by the protected application—this is especially true for larger and more complex applications. Consequently, depending on the functionality of the shellcode or ROP payload and the needs of the targeted application, code stripping may not be able to block some exploits. Indeed, as we later show in our experimental evaluation (second column of Table 2) the number of remaining *critical* system functions after code stripping for the same set of applications is quite high. For the purposes of this work, we consider as critical a set of 51 functions from `kernel32.dll`, `ws2_32.dll`, `wininet.dll`, `msvcrt.dll`, `urlmon.dll`, and `ntdll.dll`, which we found in use by a collection of 251 shellcode and 30 ROP payload samples (about which more details are provided in Section 5). Our set is very similar to the set of critical API functions protected by other runtime exploit prevention systems like ROPGuard [29] and kBouncer [55].

Based on the above, we are motivated to explore whether *further* attack surface reduction is possible by *neutralizing parts of the logic* of the critical functions that cannot be removed by code stripping. As we show in the rest of this paper, the needs of malicious code and user applications when it comes to those remaining critical functions are quite divergent, allowing us to enforce strict policies that break the functionality of exploit code without affecting the normal operation of the protected application.

Threat Model

Our aim is to protect user applications that may contain exploitable memory corruption vulnerabilities. We assume that the attacker is able to hijack the control flow of the process and execute malicious ROP code (and possibly second-stage shellcode), which interacts with the OS to achieve the attacker's end goal (e.g., remote control, DLL injection, malware installation). In that sense, data-only attacks (e.g., modifying a user authentication variable in memory [17]), which do not result in the invocation of system calls, are out of the scope of this work.

We also assume that common exploit mitigations, such as non-executable memory and ASLR, have been deployed on the system. Although Shredder offers the same defense capabilities irrespective of these defenses, as a defense-in-depth approach, it is mostly meant to be used in combination with other defenses to collectively raise the bar for successful exploitation, and prevent circumvention. As a defense based on API hooking, Shredder's policy checks must be protected so that an attacker cannot simply bypass them (e.g., by jumping over the check, or even invoking system calls directly). This can be achieved through various means, such as API checkpointing [55] or CFI [2]—we discuss in detail such possible safeguards in Section 4.2.

3 API SPECIALIZATION

The main idea behind API specialization is to create application-specific versions of the critical system API functions that a given application needs to use (and which cannot be simply removed by code stripping [50]). Especially in Windows, many critical system API functions have rich and complex interfaces that allow for increased versatility in carrying out a broad set of operations. By restricting this interface to the absolutely necessary functionality, and neutralizing any provided capabilities that are not needed, API specialization can be an effective last-resort defense for blocking crucial system interactions that exploit code needs to perform.

The specialization of a given API function requires a thorough analysis of *how* that function is used by the application, i.e., what are its expected argument values across *all* possible invocations. This information is then used to derive a *policy* that will be enforced

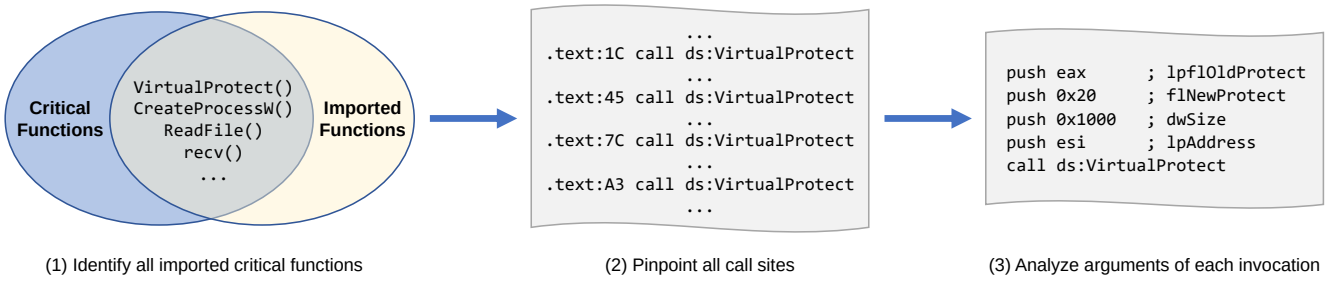

Figure 1: Overview of the offline preprocessing phase. After pinpointing the call sites of all critical API functions in the binary, each call site is analyzed using backwards data flow analysis to derive its argument values.

at runtime to block any unanticipated invocations. Although there already exist ways to manually define similar policies at the system call level (e.g., using `seccomp` or `seccomp-bpf` in Linux), our goal is to develop an *automated* solution that derives more *fine-grained* whitelisting policies at the function argument level for *closed-source* Windows applications—our focus on Windows stems from the fact that it is among the most highly targeted platforms.

Our broader goal is to develop a *practical* defense-in-depth tool for the protection of closed-source software. Although policy extraction could be performed using more sophisticated program analysis techniques at the compiler level, our assumption is that no source code or debug symbols are available, and that no modifications (e.g., binary instrumentation) can be made to the application—the latter is an important requirement for achieving deployment transparency and ensuring interoperability with existing exploit mitigations without causing compatibility problems. In that sense, Shredder follows a similar deployment model as existing hardening toolkits, such as Microsoft's EMET [49].

The process of protecting a given application comprises of two steps. In the initial offline phase, the application is statically analyzed to derive the specialization policy for the critical API functions used. After the policies are derived, a runtime enforcement module intercepts each critical function invocation and checks if it violates the policy for the given function. If a critical function is invoked using an unanticipated combination of arguments, then the system terminates the process and notifies the user.

3.1 Offline Policy Generation

To protect an application, Shredder first statically analyzes its main executable and any associated modules to identify the superset of critical system API functions used, and then derives a specialization policy for each one of them. This offline process must be performed only once per application, and the extracted policies are stored in a separate file on disk for future executions.

Figure 1 illustrates the main steps of the offline preprocessing phase. After identifying the set of imported critical functions, the binary is scanned to pinpoint all call sites of each function. Although accurately and fully disassembling the code of Windows binaries is a challenging problem [5], fully accurate disassembly is not needed for our purposes. As we focus only on the call sites of *imported* functions, these can be *accurately* pinpointed by looking for code references to the respective entries of the import table. Having the control flow instruction of each call site as a trusted reference, we can then perform backwards analysis to accurately extract the code (within the function that contains the call site) that is involved in the preparation of the arguments. For each function argument, we then attempt to derive its possible values by performing interprocedural backwards data flow analysis. In contrast to pinpointing call sites, this later analysis stage *is* subject to disassembly and interprocedural static code analysis inaccuracies (as ideally complete control flow graph information is needed), and is thus performed in a conservative, best-effort way.

There are three main types of arguments we have to deal with: i) constant values, ii) stack variables, and iii) register values. Note that we care about obtaining the argument values, and not the way they are passed to the callee (i.e., through the stack in 32-bit systems, or through registers in 64-bit systems). Given that our data flow analysis handles propagation through both registers and memory, it can support both 32-bit and 64-bit executables.

For stack variables, the callee typically accesses passed arguments and local variables in relation to the frame pointer, and thus we use a similar method to find their values, by tracing back to the caller function and identifying any statically derived pushed arguments. Register values are usually the result of arithmetic or logic operations that may involve other registers or memory locations. Our backwards data flow analysis can derive a set of possible values for a given register using a lightweight form of symbolic execution, as long as they originate from static data or constant values.

When the backwards data flow analysis reaches the beginning of the function that contains the call site being analyzed, it moves on to its previous caller(s) as long as the relevant control flow graph information is available. As we discuss in Section 4, due to the exponential growth of backwards control flow paths, in our current implementation we have set a conservative depth of three functions to keep the analysis time short. Obviously, in many cases it is impossible to derive the values of some arguments, either because these will only be determined at runtime (e.g., file names, memory addresses, process IDs), or due to the limitations of our control and data flow analysis. On the other hand, many argument values are derived from static data that is quite easily reachable through data flow analysis (e.g., access modes, memory protection flags, allocation sizes, network connection parameters, hard-coded names). In fact, as we show in Section 5, we found that it is possible to extract "known" values for more than half of the arguments of

the analyzed critical functions. More importantly, many of these arguments have clearly disparate value sets for the needs of benign and malicious code, allowing us to derive effective policies.

After the end of our analysis for the whole program, a given argument is classified as either *known* or *unknown*. Known arguments belong to one of the following types:

(1) *Flag Argument*: Each flag value has a special meaning, and multiple flags can be combined through a logic OR operation.
(2) *Range Argument*: A numeric range specifies an upper and lower limit on the expected values (these often correspond to memory address ranges).
(3) *Distinct Value Argument*: A set of possible distinct values that this argument may take (e.g., numbers or strings). Arguments such as allocation sizes (of which we observe only a few distinct values) often fall into this category.

An argument is considered unknown, if a constant value cannot be determined for *each and every call site* of its function. We follow this conservative approach despite the fact that we can often determine the values of a given argument for the majority of a function's call sites, and just miss a few. In such cases, API specialization is coupled with CFI to restrict further the set of attacker-controllable invocation points, by enforcing call-site-specific policies. Although our prototype currently supports call-site-specific policies (the design and implementation of which we present in the appendix), given that only a few Windows applications are currently CFI-enabled, we leave the evaluation of such a scheme as part of our future work. As we demonstrate in Section 5, our current more coarse-grained approach using a single program-wide policy per function is still quite effective in thwarting real exploits.

In some cases, even if some arguments are unknown, they can still take part in policy specification by considering relationships across two or more arguments that are reflected in the argument preparation code of call sites. For instance, from an instruction sequence like `push eax; push eax; push ebx; call <function>`, we can infer that the first two arguments have the same value, even if the actual value is unknown. If this invariant holds for all call sites of a given function, then it can be captured as part of the enforced policies for that function.

Example policies generated by our backward data flow analysis algorithm are:

(1) `VirtualProtect(): (arg3 = 0x20 AND arg2 = 0x1000) OR (arg3 = 0x104)`
(2) `VirtualAlloc(): (arg1 == arg2)`

3.2 Runtime Policy Enforcement

After the preprocessing phase is completed, the resulting policies are ready to be used for runtime protection. Shredder relies on library interposition to check the arguments of a sensitive API function invocation before permitting it to proceed. A call is permitted to proceed only if its arguments are in accordance with the policy for the given function. If not, the call is dropped.

Although we assume that code stripping [50] has already been applied to the target application, even if this is not the case, Shredder does partly provide equivalent protection by also blocking the invocation of any function that is not present in the import table of the protected application. This provides the same level of protection when it comes to whole-function reuse, but does not help with ROP gadget reuse from the code sections of the non-imported functions, as their code is still present in the address space of the process—this code removal feature could of course be incorporated into Shredder with some engineering effort, as we further discuss in Section 6.

4 IMPLEMENTATION

To demonstrate the effectiveness of API specialization, we have developed Shredder, a prototype protection tool for Windows PE binaries (both main executables and dynamic link libraries). Shredder has been developed and tested on the 64-bit version of Windows 10, and consists of two main components: an offline static policy extractor, and a user-space thin interposition layer between applications and Windows API functions. We have also developed a shellcode and ROP payload analysis framework to evaluate the effectiveness of the derived policies in blocking existing exploits.

4.1 Static Policy Extraction

Our policy extraction pass has been implemented on top of the IDA Pro disassembler through its IDAPython scripting environment. Our IDAPython tool reads each input executable and initially scans it to find all call sites of critical API functions (we assume that the complete set of an application's modules is available in advance for analysis). Note that call sites can be accurately pinpointed even without any debug or symbolic information (e.g., PDB files), by identifying the respective references through the import table.

For each identified call site, the tool then performs backwards inter-procedural data flow analysis to derive the function argument values used in that particular call site. Due to the inherent imprecision of code disassembly and control flow graph extraction, unfortunately we cannot rely on existing sophisticated data flow analysis frameworks that operate at the source code or intermediate representation (IR) layers. Although we initially explored the use of IR-lifting tools (e.g., McSema [1]) that would enable data flow analysis at the IR level, we found that the analysis results were not significantly better (from a policy extraction perspective) for our purposes, compared to our custom, argument-focused data flow analysis implementation.

Of particular usefulness is IDA pro's *stack variables window* data structure, which contains local variables and function arguments extracted during static analysis. If a value is derived by an input from a previous function, we leverage the computed control flow graph to perform backwards data flow tracking and attempt to identify its source. For our current implementation, we have set a limit of three functions for the recursion depth to keep analysis time low. Based on our experimentation, higher recursion values provided only diminishing returns.

If the value of an argument is not found within these iterations, the value is considered unknown. As reported in Section 5, despite the best-effort nature of our approach and the limitations of our data flow analysis, it still achieves good coverage for the particular arguments of critical functions, providing high discriminatory capacity between benign and malicious code.

4.2 Runtime Interposition Layer

Our current implementation uses Microsoft Detours [30] framework to selectively intercept protected API calls. During initialization, the extracted policies are loaded from its respective file, and are verified at runtime against each intercepted invocation. If there is no policy violation, the actual API function is then called, otherwise an alert message is displayed and the process then terminates. Due to the conservative nature of our policy extraction phase, there is no false positive issue, as the policies rely only on statically-derived argument values observed through the whole application code.

As any other defense based on function hooking, if no other precautions are taken, attack code could bypass Shredder's policy check by either jumping over Detour's hook, or by invoking the respective system call directly (e.g., through `ntdll.dll`).[1] In the context of code-reuse attacks, which is our primary focus, this issue can be addressed by ensuring that system calls can be solely invoked through Windows API functions, and not directly. This can be achieved in several ways. If the protected application employs CFI, then the enforced CFI policy will prohibit arbitrary control transfers to the middle of API functions or even straight to system call wrapper functions.

Even if CFI is not an option, previous systems (e.g., kBouncer [55]) have proposed a checkpointing mechanism that relies on the kernel to perform the policy enforcement at the entry point of the API function, and set a checkpoint that is checked upon system call entry. Tying the checkpointing code *after* the policy check ensures that exploit code cannot fake a checkpoint without violating the enforced policy. We refer the interested reader to description of this mechanism by Pappas et al. [55] for more details. More lightweight hook protection mechanisms, such as those employed by Microsoft's EMET [49], could also be employed. As hooking protection is an orthogonal issue for which solutions already exist, we have left its implementation as part of our future work.

5 EVALUATION

In this section, we present the results of our experimental evaluation of Shredder in terms of runtime overhead and effectiveness against real-world exploits. All experiments were performed on a system equipped with an Intel Core i7-5650U CPU, 8GB RAM, 256GB SSD, running the 64-bit version of Windows 10 Education.

5.1 Data Set

To evaluate the effectiveness of Shredder against real-world exploits, we gathered a diverse set of 251 shellcode and 30 ROP code samples from Metasploit, Exploit DB, and individual real-world and proof-of-concept exploits. We consider both shellcode and ROP payload samples, as in modern exploits ROP code is typically used just to enable the execution of a second-stage shellcode, by giving it execute memory permission. The shellcode then performs further malicious activities, and typically invokes many more system API functions. However, although ROP code is much more complex to construct, sophisticated exploits may avoid the use of second-stage shellcode altogether, and implement the whole functionality using reused code [9]. In either case, the exploit code unavoidably has to invoke several system API calls. To get a better insight about which stage Shredder is more effective in blocking, we present our results separately for shellcode and ROP payloads.

To evaluate the effectiveness of our policy extraction against these payloads, we use a set of 10 popular Windows applications that are typically targeted by in-the-wild attacks. As the choice of payload used in a given exploit depends on the attacker's goal, the characteristics of the vulnerability, and the protections deployed on the victim system, we make the conservative assumption that any shellcode or ROP payload from our data set can be potentially used as part of exploits against any of the considered applications. That is, we do not make any assumptions about the actual functionality of the exploit code (e.g., creating sockets, writing files, allocating memory), but conservatively assume that *any* of these operations may be performed. Similarly, only a subset of ROP payloads will be applicable for a given application, depending on the available gadgets in its code image. Still, we assume that all ROP payloads may be applicable, as we care only about the API functions they use, and not about the specific gadgets they rely on to invoke them. Full details and sources for the particular ROP payloads used in our evaluation are provided in Section B and Table 6 in the appendix.

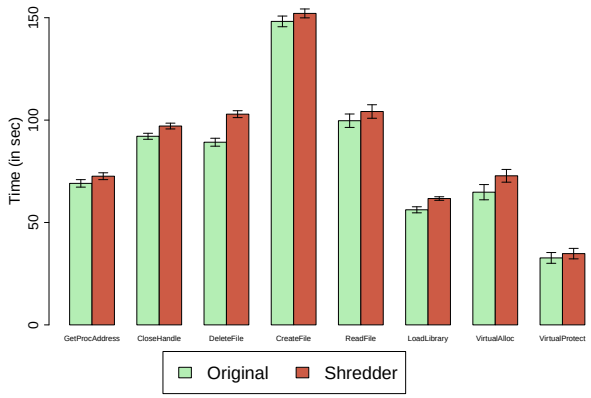

Figure 2: Worst-case runtime overhead of Shredder's runtime policy enforcement, for artificial stress-test cases that repeatedly invoke critical API functions.

5.2 Runtime Overhead

Measuring the performance impact of defenses based on API hooking on real world applications like media players, text editors, and web browsers, i.e., such as the ones we have included in our set, is a challenging task due to their interactive nature. Instead, to measure the runtime overhead of Shredder's API call interception and policy checking, we used a set of custom programs that *repeatedly* invoke critical (hooked) API functions with various sets of argument values and enforced policies—although this is a worst-case behavior that is not encountered in real-world applications, it provides an upper bound on what can be expected in practice.

[1] Although direct system call invocation is much more challenging in Windows compared to Linux [64], this is still an option for highly targeted exploit code constructed for a particular victim system [3].

To measure API call overhead, each invocation is performed with a different set of arguments, a check is made against 10 different policies (a larger number than needed in practice), and each test case invokes a given function 100 million times, which is orders of magnitude more than the average number of calls made by the applications we tested for typical workloads. As shown in Figure 2, for all tested API functions, the total execution time when using Shredder compared to the original test cases is only slightly increased.

Using further micro-benchmarks, we found that the per-call overhead on average is less than 20ns. We also applied Shredder on the SPEC2006 benchmarks, and found no measurable overhead. The main reason is that the benchmarks are designed for CPU performance evaluation, and thus invoke very few API calls that need protection (`ExitProcess()`, `WriteFile()`, `ReadFile()`, `CloseHandle()`, `CreateFileW()`).

5.3 Policy Generation

The entire premise of Shredder's protection is that we can often differentiate between legitimate and malicious invocations of critical API functions, due to the disparate set of argument values used in each case. To evaluate the extent to which Shredder is capable of generating restrictive policies regarding the expected values of critical API function arguments, we used a set of 10 popular—and often targeted by in-the-wild exploits—Windows applications, listed in Table 2. The results of this section focus only on the *remaining* critical API functions after the application of code stripping [50], the arguments of which Shredder can restrict further.

5.3.1 Analysis Time. Each application is first analyzed with IDA Pro using its standard code disassembly and control flow extraction settings (we refrain from using any optional more aggressive disassembly heuristics). Then Shredder's offline analysis phase takes place to identify all critical API function call sites, and perform backwards data flow analysis to derive their known argument values. As shown in the second and third columns of Table 2, the number of imported critical functions and respective call sites is typically small, with iTunes and Firefox having the highest number of critical functions. Due to the low number of call sites that need to be analyzed, Shredder's data flow analysis takes only a few seconds, as shown in the fourth column (the reported time excludes the time IDA Pro needs for code disassembly).

5.3.2 Function Argument Identification. Depending on the accuracy of the analysis, the nature of the arguments used, and the complexity of the application, the number of extracted policies varies across different applications, and roughly grows linearly with the number of critical functions, as shown in the fifth column of Table 2. The highest number of policies (61) is derived for Notepad++, which involve 11 functions (as shown in the sixth column). On the other hand, only five policies could be generated for VLC, which though still prevent the majority of shellcode samples and some of the ROP payloads. This is a worst-case scenario, as VLC has a few invocations of `VirtualProtect()` performed with the executable memory flag set, which prohibit the extraction of any meaningful policy (i.e., all possible values for that argument may be legitimately observed).

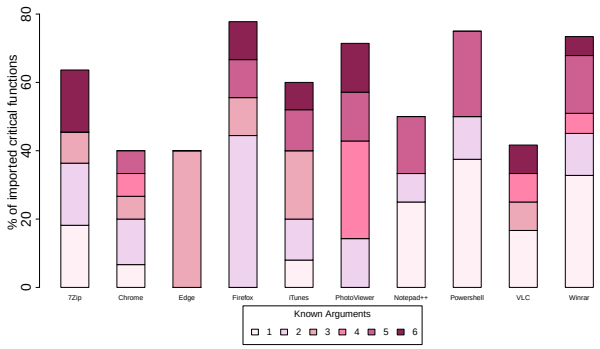

Figure 3: Percentage of critical API functions with at least one known argument, for the tested applications. The breakdown in each bar corresponds to the actual number of known arguments.

To get a better understanding about the cases in which Shredder fails to generate a policy, Table 3 lists the functions for which a policy could not be generated, ordered according to their popularity in terms of number of call sites (third column) across all applications. We see that the most frequently challenging functions are `CloseHandle()` and `DuplicateHandle()`, which both take as argument a handle to an object, which can only be determined at runtime, followed by file-related functions, which also usually involve non-static arguments. Although all arguments remain unknown (last column) in all call sites of these two functions, this does not impact Shredder's effectiveness significantly, as these functions cannot have any significant security impact on their own, and are mostly used in combination with other API functions in some shellcode samples. On the other hand, for some truly critical functions like `VirtualProtect()`, Shredder can still derive known arguments for the majority of call sites—but not for all of them, hence the inability to derive a policy. In such cases, more fine-grained per-call-site policies can be used to force the attacker to use only the callsites with unknown argument instances, as discussed in Appendix A.

Fortunately, for most functions, Shredder *is* able to derive *all possible values* for a given argument (in which case it becomes a *known* argument, as discussed in Section 3.1). Figure 3 shows the percentage of critical functions in each application with at least one known argument. In addition, each bar shows the breakdown in terms of the number of arguments with known values. For instance, in 7zip, about 20% of the imported critical functions have one known argument, 20% have two known arguments, and there are also a few functions with three and six known arguments. Overall, with the exception of Chrome, Edge and VLC, Shredder is able to derive known argument values for more than half of the critical functions. Usually, a larger number of known arguments for a given function leads to a larger number of policies, as there is a higher potential for deriving further restrictions across different argument combinations, as discussed in Section 3.1.

Table 2: Policy generation and protection effectiveness results for a set of Windows 64-bit applications.

Application	Critical Functions	Call Sites	Analysis Time (sec)	Generated Policies	Functions w/ Policies	Broken Exploit Payloads		
						Shellcode	ROP	ROP Equiv.
7Zip	8	39	1.2	14	7	242 (96%)	30 (100%)	30 (100%)
Google Chrome	14	113	3.4	29	9	251 (100%)	30 (100%)	30 (100%)
Microsoft Edge	15	337	4.8	28	12	248 (98%)	30 (100%)	30 (100%)
Mozilla Firefox	20	104	1.8	54	17	244 (97%)	30 (100%)	30 (100%)
iTunes	33	315	2.3	59	25	246 (98%)	22 (65%)	1 (3%)
PhotoViewer	8	17	1.4	21	6	228 (91%)	30 (100%)	30 (100%)
Notepad++	12	84	1.1	61	11	244 (97%)	30 (100%)	30 (100%)
Powershell	5	5	0.4	14	4	246 (98%)	30 (100%)	30 (100%)
VLC	12	24	0.5	11	5	216 (86%)	9 (30%)	1 (3%)
Winrar	10	120	1.3	34	8	242 (96%)	30 (100%)	30 (100%)

Table 3: Functions for which Shredder cannot derive a policy in some applications.

API Function	Apps	Call Sites	Known Instances	Unknown Instances
`CloseHandle()`	10	124	0	124
`DuplicateHandle()`	7	13	0	13
`DeleteFileW()`	6	12	0	12
`ReadFile()`	2	3	1	2
`WriteFile()`	1	1	0	1
`VirtualProtect()`	5	6	3	3
`ExitThread()`	4	6	1	5
`ioctlsocket()`	1	3	2	1
`CreateProcessW()`	1	5	4	1
`InternetReadFile()`	2	4	3	1
`InternetOpenW()`	2	4	2	2
`ExitProcess()`	3	3	0	3
`bind()`	1	1	0	1
`closesocket()`	1	1	0	1

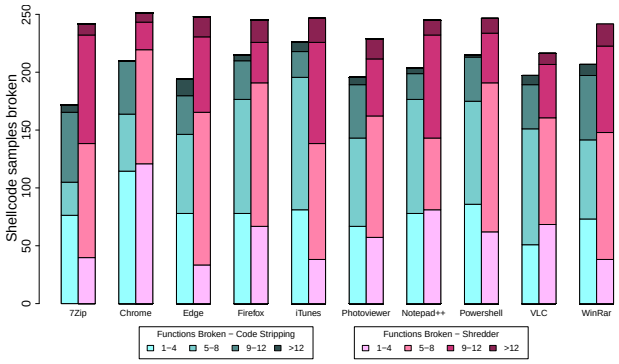

Figure 4: Number of shellcode samples broken by code stripping [50] and Shredder. Bar segments denote the distribution of broken critical API functions in each case.

5.4 Protection Effectiveness

After generating API specialization policies for each application, we set out to explore their effectiveness against real-world exploits, and the added protection benefit compared to code stripping [50], i.e., simply removing any non-imported system API functions.

5.4.1 Effectiveness Against Real-world Exploits. Based on our conservative assumption that any of the 251 shellcode and 30 ROP payload samples (for brevity, we collectively refer to both as "payloads" in the rest of this section) in our data set can be used against any application, we used our custom payload analysis framework to run each sample and capture all critical API function invocations. We then compare these invocation patterns with the API specialization policies of each application to determine whether Shredder is able to block a given payload. A payload is *broken* if at least one of the API functions it uses violates one the enforced policy. This means that either the function is not imported and used at all (in which case code stripping alone would also block it), or that all invocations of the function use unanticipated argument values, compared to the set of *known* arguments that Shredder expects.

As shown in Table 2 (columns "shellcode" and "ROP"), Shredder is able to break 90–100% of the payloads for most applications. Especially for ROP payloads, the only cases Shredder is not able to break all of them are iTunes and VLC. Upon further inspection, this is because: for iTunes and Reader, there are `VirtualAlloc()` call sites that set the executable memory flag, and for VLC, there are four `VirtualProtect()` call sites, two of which set the executable memory flag, and two other in which the arguments are unknown.

5.4.2 Comparison with Code Stripping. We compare the added benefit of Shredder over code stripping [50] i.e., just removing any non-imported API functions, by repeating the same experiment, this time without enforcing any policies to the remaining (imported) critical API functions. Code stripping alone can break payloads by prohibiting the use of critical API functions that are needed by the payload but not by the application.

Figures 4 and 5 show the number of broken shellcode and ROP payloads, respectively, for code stripping and Shredder. The breakdown in each bar of Figure 4 denotes the number of functions broken in each case (as shellcode typically uses several API calls).

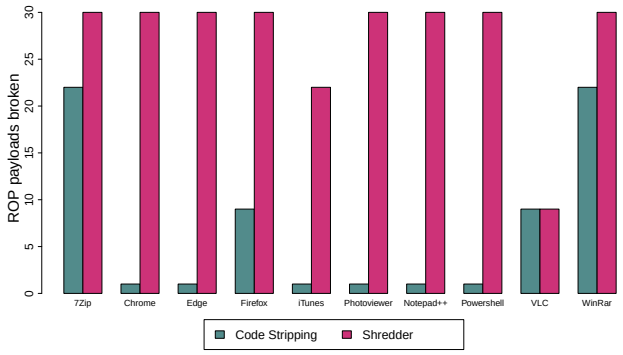

Figure 5: Number of ROP payloads broken by code stripping [50] and Shredder for 64-bit applications. Shredder breaks all ROP payloads for eight out of the 10 applications.

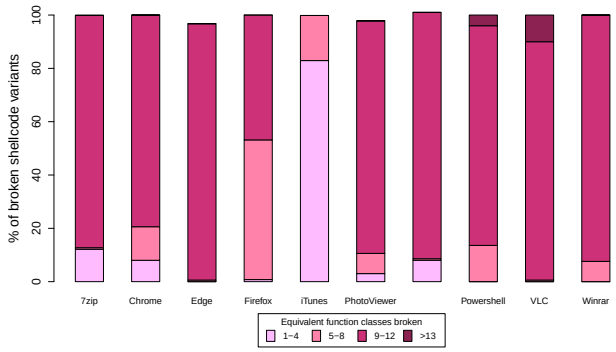

Figure 6: Percentage of equivalent shellcode variants broken by Shredder for 64-bit applications. Bar segments denote the distribution of broken function classes in each case.

For instance, in the case of Chrome, Shredder blocks all 251 shellcode samples by breaking up to 14 functions (although the first invalid invocation will lead to process termination, here we want assess how comprehensive the derived policies are). In contrast, code stripping blocks only about 70% of the shellcode samples. In all cases Shredder offers a significant benefit over code stripping.

When it comes to ROP payloads, the benefits of Shredder are even clearer. Memory-related functions like `VirtualAlloc()` and `VirtualProtect()` are typically used by both legitimate applications and exploit code, and thus in most cases code stripping cannot remove them. In contrast, Shredder blocks *all* 30 ROP payloads in eight out of the ten applications, 22 of them in one application, and nine in case of VLC (for reasons we explained earlier). Overall, it is clear that API specialization achieves better protection than code stripping alone for both types of payloads, and especially for ROP payloads, which have become indispensable for modern exploits.

5.4.3 Robustness to Circumvention Attempts. The fact that Shredder blocks most of the tested payloads in the context of the given applications does not mean that attackers cannot modify their shellcode or ROP code so that it conforms to the enforced API specialization policy. Knowing that Shredder is in place, an attacker could pick a different set of API functions to achieve the same goal. Although assessing this possibility in general is a challenging task, due to the multiple combinations of API calls that an attacker could use for a given task, we attempted to explore it by considering classes of equivalent functions (or function combinations) that can achieve the same goal.

For ROP payloads, we focus on functions that aim to give execute permission to a second-stage shellcode. Given that there are two main ways to achieve this, we consider the equivalence of `VirtualAlloc()` and `VirtualProtect()`, which (with adequate code restructuring) could be used interchangeably. Consequently, if for a given application only one of them is blocked, then attackers could use the other one in their payloads to circumvent Shredder. To assess the robustness of Shredder against such a circumvention attempt, we repeated our evaluation by making the conservative assumption that both functions can be used interchangeably in the 30 ROP payloads. As shown in the rightmost column of Table 2, even under these unfavorable circumstances, Shredder can still block all ROP payloads for eight applications. For the rest two, it manages to block only the single ROP payload that relies on `WinExec()`, and the other 29 payloads become usable, as Shredder has managed to derive policies only for one of the two memory-related functions in those applications. Despite its best-effort nature, Shredder still meaningfully raises the bar against exploitation, as in many cases it considerably restricts the options an attacker has for the construction of the ROP payload.

For the set of 251 shellcode samples, we derived 14 sets of equivalent functions for different types of generic functionality. For example, allocating memory with `VirtualAlloc()` could be replaced by `heapAlloc()`, `globalAlloc()`, `localAlloc()`, `malloc()`, or `new()`. A detailed list of all such equivalence classes is provided in Table 7 in the appendix. Given these classes, we create all possible variants of the 251 shellcode samples in our data set, totalling more than 23 million samples. Figure 6 shows the percentage of shellcode variants that Shredder can still block. For most applications, the numbers are approaching 100%. This stems from the fact that many of the equivalent functions are rarely used, or when used, they are restricted by Shredder's policies.

5.4.4 Real-world Use Case. To demonstrate the benefits of the attack surface reduction that API specialization offers, we evaluated Shredder using a real-world exploit against Chakra, the JavaScript engine used in Microsoft Edge. The exploit [43] takes advantage of two vulnerabilities (CVE-2016-7200 and CVE-2016-7201) to gain code execution on Windows 10, while bypassing *all* exploit mitigations incorporated into Edge (at the time the exploit was written), including: heap spray disruptors, by precisely allocating a minimal amount of memory; ASLR, by leveraging a memory disclosure vulnerability; export address table filtering (EAF+), by using hardcoded addresses for critical API functions instead of dynamically looking them up; and control flow guard (CFG), as well as gadget chain detectors like kBouncer [55] and ROPGuard [29], by using *only two* gadgets to set up a `VirtualProtect()` call, which is made through a *legitimate* wrapper function—conforming this way to the legitimate control flow graph enforced by CFG's CFI policy and avoiding any stack disruption.

We verified that the above exploit works correctly against Edge v40.17017.0.0 on Windows 10, and then tested it again with Shredder enabled. When the ROP code invoked `VirtualProtect()` to give execute permission to the second-stage shellcode, Shredder reported a policy violation and terminated the Edge process. Although all previous exploit mitigations were bypassed, API specialization helped as a last-resort measure to block the exploit.

6 LIMITATIONS AND FUTURE WORK

Shredder's conservative approach, even when call-site-specific policies are used, is not able to prevent all attacks. This stems from the fact that if there is even one call site that cannot be restricted, then attackers are likely to be able to use it with adequate effort. However, as all exploit mitigations, Shredder's goal is not to provide a silver bullet solution, but to deprive attackers from the current unrestricted convenience of using any API function they will, in any way possible. In doing so, Shredder does not introduce any measurable overhead, and is fully compatible with existing applications and exploit mitigations. Consequently, it should be viewed as a step towards improving the security of a system by limiting the "latent complexity" that works in favor of attackers [25].

A limitation of our current prototype is that it simply relies on Microsoft's Detours [30] framework for library interposition. As discussed in Section 4.2, this issue can be addressed with adequate engineering effort by employing a secure function hooking technique, such as the checkpoint-based approach used in kBouncer [55].

In contrast to code stripping [50], Shredder currently does not actually remove any code from the address space of the protected process, but just restricts the use of i) non-imported API functions (similarly to code stripping), and ii) remaining (i.e., imported) critical functions, according to the derived policies. Although identifying and removing all code dependencies of a given unused function (including any non-exported internal functions that are not needed anymore) is already a challenging problem when source code is not available [50], in principle, program slicing techniques could be used to remove parts of unnecessary code from within remaining functions, according to the derived policies. We leave such more fine-grained code removal techniques as part of our future work.

7 RELATED WORK

As non-executable memory protections and address space layout randomization (ASLR) are not enough for the prevention of modern ROP exploits (due to the proliferation of memory disclosure vulnerabilities [42, 65]), there has been active research on a wide variety of additional defenses. Two main approaches that we can identify include static protection and runtime monitoring techniques.

Approaches of the former type include i) compiler-level techniques for applying control flow integrity (CFI) [2], enforcing the integrity of code pointers and the stack [41], or protecting indirect control transfers [45, 53], and ii) binary-level techniques for applying code diversification [54, 72] or various forms of CFI [75, 76]. Runtime monitoring approaches augment the execution of a process at various levels (e.g., instruction, system call) to prevent attacks using various techniques, such as performing anomaly detection by checking for an unusually high frequency of `ret` instructions [16, 22], ensuring the integrity of the stack [23], randomizing the locations of code fragments [35], or preventing illegal indirect transfers [19, 55]. In this section, we focus on the areas of API-level monitoring and code surface reduction, which are more closely related to our work.

7.1 API-level Monitoring

Monitoring execution at the system call or API level strikes a good balance in terms of performance (system call or API function invocations are infrequent, e.g., compared to monitoring at the instruction level) and analysis accuracy (given that malicious code has to eventually interact with the OS). Consequently, similar to Shredder, many previous defenses rely on system call or API call interception to perform various types of checks in order to block the execution of malicious code.

In the front of defending against return-oriented programming (ROP) exploits, several approaches rely on the idea of performing runtime checks to identify control flow abnormalities that usually appear when ROP code is executing. Given that checking all control flow transfers at runtime introduces a very high performance overhead, systems like kBouncer [55] and ROPGuard [29] perform these checks only before the execution of critical API functions.

In particular, kBouncer [55] relies on the Last Branch Record (LBR) feature of recent processors to inspect the sequence of indirect branch instructions that led to the intercepted API function invocation. Similarly, ROPGuard [29] performs a variety of checks, such as validating whether the return address to the caller function points to a call-preceded instruction, and checking whether the stack pointer falls within the boundaries of the actual stack. Many of ROPGuard's checks have been incorporated to Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [49], which also implements many of its exploit mitigation technologies by intercepting critical API calls.

Several years before EMET, the security community used similar approaches to build protections for Windows systems by enforcing policies or implementing detection heuristics at the system API level. WHIPS [8] is a host-based intrusion detection system (HIDS) for Windows 2000/XP/2003 that enforces rules kept in an access control database by intercepting Native API calls. The creation of the enforced rules is an orthogonal issue not addressed in that work, which the authors mention, can be performed either in a manual or automatic manner. Anderson et al. [4] implemented a host-based code injection attack detector by using Detours to intercept and inspect network inputs for the presence of an excessive number of NOP instructions, which frequently precede the shellcode.

Similar systems were prototyped for Linux even earlier. For instance, the REMUS system [10] implements a reference monitor for system call invocations as a loadable Linux kernel module. Libsafe and Libverify [7] aim to transparently prevent buffer overflow exploits by enforcing buffer sizes and verifying return addresses on the stack through library interposition. Many other systems rely on system call interposition to enforce blacklisting or whitelisting policies [31, 33, 56]. Numerous other works in the span of more than two decades have proposed systems that rely on system call interposition to defend against intrusions using anomaly detection [11, 27, 28, 37, 46, 62, 71, 74].

7.2 Code Surface Reduction

Several works have started exploring the concept of "slimming down" the code surface of applications, by removing any unnecessary code. This overall strategy has the dual goal of reducing the threat of exploitation by removing any vulnerabilities that were present in the unused code, and make exploit construction harder (especially for code reuse), by reducing the potential functions or ROP gadgets that could be used by an attacker.

Software winnowing [48] is one such approach that applies this concept to specialize the code of applications and libraries. The authors have implemented a code specialization tool on top of LLVM, called OCCAM, (Object Culling and Concretization for Assurance Maximization), which generates specialized versions of applications according to a given configuration or deployment context. OCCAM supports both intra-module and inter-module winnowing, and can perform sophisticated code specialization by taking into account all program dependencies. Piecewise Debloating [58] is another approach for debloating libraries and main executables. At compilation and link time, the framework collects accurate control flow graph information, which is embedded into the resulting binaries. At run time, this information is used to load only the relevant portions of code to the memory. The unused portions are replaced by illegal instructions thus shrinking the attack surface.

Shredder employs some similar ideas, but as we have already discussed, the constraint of operating transparently on closed-source applications severely limits the types of program analysis that we can rely on. Furthermore, it is complementary to the above approaches, as i) it moves one step further, by debloating further the *remaining* code, in terms of what part of function interfaces remains operational, and ii) focuses on system API libraries, instead of a program's own modules.

Similar code surface reduction approaches have been proposed for the Linux kernel, the multi-purpose nature of which makes the underlying code base immense. Kurmus et al. [38] implemented kRazor, a system that limits the amount of kernel code accessible to an application. In a training phase, the system uses dynamic instrumentation of all the kernel functions to collect the set of used functions under certain workloads. In the enforcement phase, the system then limits usage to that set of functions. Similar methods have been employed by other systems [39, 40, 68] to create custom minimized kernels suited for specific workloads, achieving a code surface reduction in the range of 50–85%. While the above systems create custom single-purpose kernels for certain workloads, FaceChange [34] uses multiple minimized kernels, one tailored to each application, which are swapped accordingly upon context switch.

8 CONCLUSION

Motivated by the concept of attack surface reduction, and by the need for practical and composable defense-in-depth mitigations that can be readily and transparently applied for the protection of applications that are targeted by in-the-wild attacks, in this work we have presented Shredder, an exploit mitigation tool for Windows programs. Shredder uses API specialization to restrict the interface of critical system API functions according to the actual needs of the protected program, and neutralize parts of their functionality that are often crucial for the operation of malicious code. The results of our experimental evaluations show that Shredder offers a significant improvement over code stripping [50], a previous code surface reduction technique for closed-source Windows applications, by blocking the execution of 18.3% more shellcode and 298% more ROP code samples, while incurring a negligible runtime overhead.

ACKNOWLEDGMENTS

We would like to thank Collin Mulliner, Azzedine Benameur, and the anonymous reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045, with additional support by Accenture. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, DARPA, or Accenture.

REFERENCES

[1] McSema: Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode. (2017). https://github.com/trailofbits/mcsema.
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*. 340–353.
[3] Alex Abramov. Manually Enumerating Process Modules. (2015). http://www.codereversing.com/blog/archives/265.
[4] S. Andersson, A. Clark, G. Mohay, B. Schatz, and J. Zimmermann. 2005. A framework for detecting network-based code injection attacks targeting Windows and UNIX. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*.
[5] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25rd USENIX Security Symposium*. 583–600.
[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. 1342–1353.
[7] Arash Baratloo, Navjot Singh, and Timothy Tsai. 2000. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*.
[8] Roberto Battistoni, Emanuele Gabrielli, and Luigi V. Mancini. 2004. A Host Intrusion Prevention System for Windows Operating Systems. In *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS)*. 352–368.
[9] James Bennett, Yichong Lin, and Thoufique Haq. The Number of the Beast. (2013). http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html.
[10] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. 2002. Remus: A Security-enhanced Operating System. *ACM Trans. Inf. Syst. Secur.* 5, 1 (Feb. 2002), 36–61.
[11] S. Bhatkar, A. Chaturvedi, and R. Sekar. 2006. Dataflow anomaly detection. In *Proceedings of the IEEE Symposium on Security & Privacy*.
[12] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *IEEE Symposium on Security and Privacy*. 243–258.
[13] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[14] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (April 2017), 33 pages. https://doi.org/10.1145/3054924
[15] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*. 161–176.
[16] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. 2009. DROP: Detecting Return-Oriented Programming Malicious Code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*. 163–177.
[17] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th*

USENIX Security Symposium.

[18] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P).*

[19] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. (2014).

[20] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P).*

[21] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[22] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC).* 49–54.

[23] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *Proceedings of the 6th ACM ASIACCS.* 40–51.

[24] Solar Designer. Getting around non-executable stack (and fix). (1997). http://seclists.org/bugtraq/1997/Aug/63.

[25] Thomas Dullien. 2018. Security, Moore's law, and the anomaly of cheap complexity. CyCon.

[26] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS).* 901–913.

[27] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong. 2003. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security & Privacy.* 62–75.

[28] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. 1996. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Security & Privacy.*

[29] Ivan Fratric. 2012. ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks. (2012). http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf/

[30] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. https://www.cs.columbia.edu/~junfeng/10fa-e6998/papers/detours.pdf.

[31] Tal Garfinkel. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[32] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY).* 325–336.

[33] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium.*

[34] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* 491–502.

[35] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P).* 571–585.

[36] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. (2005). http://www.suse.de/~krahmer/no-nx.pdf.

[37] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. 2003. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS).* 326–343.

[38] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. 2014. Quantifiable Run-Time Kernel Attack Surface Reduction. In *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* 212–234.

[39] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. 2011. Attack Surface Reduction for Commodity OS Kernels: Trimmed Garden Plants May Attract Less Bugs. In *Proceedings of the 4th European Workshop on System Security (EuroSec).*

[40] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[41] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI).* 147–163.

[42] MWR Labs. MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit. (2013). https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit/.

[43] Aaron Lamb. The Chakra Exploit And The Limitations Of Modern Cyber Security Threat Mitigation Techniques. https://www.endgame.com/blog/technical-blog/chakra-exploit-and-limitations-modern-mitigation-techniques. (2017).

[44] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security & Privacy.* 276–291.

[45] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating return-oriented rootkits with "Return-Less" kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys).* 195–208.

[46] P. Li, H. Park, D. Gao, and J. Fu. 2008. Bridging the Gap between Data-Flow and Control-Flow Analysis for Anomaly Detection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* 392–401.

[47] M. Russinovich, D. A. Solomon, and A. Ionescu. 2012. Windows Internals. (2012).

[48] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated Software Winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15).* 1504–1511.

[49] Microsoft. Enhanced Mitigation Experience Toolkit. ([n. d.]). http://www.microsoft.com/emet.

[50] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. Black Hat USA.

[51] Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. *Phrack* 11, 58 (Dec. 2001).

[52] Tim Newsham. Non-exec stack. (2000). http://seclists.org/bugtraq/2000/May/90.

[53] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC).* 49–58.

[54] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy.* 601–615.

[55] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium.* 447–462.

[56] Mathias Payer and Thomas R. Gross. 2011. Fine-grained User-space Security Through Virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).* 157–168.

[57] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the 12th European conference on Computer Systems (EuroSys).* 420–436.

[58] Anh Quach, Aravind Prakash, and Lok Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium.*

[59] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Christopher Liebchen Stephen Crane, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[60] Mark Russinovich. Inside Native Applications. (Nov. 2006). http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx.

[61] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P).* 745–762.

[62] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security & Privacy.* 144–155.

[63] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS).* 552–561.

[64] Skape. Understanding Windows Shellcode. (2003). http://www.hick.org/code/skape/papers/win32-shellcode.pdf.

[65] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P).* 574–588.

[66] Kevin Z. Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. 2016. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P).* 954–968.

[67] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. 256–267.
[68] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-time Configurability. In *Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep)*.
[69] PaX Team. Address Space Layout Randomization. (2003). http://pax.grsecurity.net/docs/aslr.txt.
[70] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1675–1689.
[71] D. Wagner and R. Dean. 2001. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security & Privacy*. 156–168.
[72] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, Zhiqiang Lin, and W Campbell Rd. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM conference on Computer and communications security (CCS)*. 157–168.
[73] Jan Werner, George Baltas, Rob Dallara, Nathan Otternes, Kevin Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*. 35–46.
[74] Andreas Wespi, Marc Dacier, and Hervé Debar. 2000. Intrusion Detection Using Variable-Length Audit Trail Patterns. In *Proceedings of the 3rd Conference in Recent Advances in Intrusion Detection (RAID)*. 110–129.
[75] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*.
[76] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*.

APPENDIX

A CALL-SITE-SPECIFIC SPECIALIZATION

In many cases, it is possible to determine the values of a given argument for the majority of a function's call sites, but miss just a few call sites in which the values for the same argument remain unknown, preventing the specification of a global policy. In such cases, Shredder's policies can be refined so that instead of having a single "weak" policy for all call sites of a given function (or no policy at all), different policies can be enforced according to the actual call site from which the function was invoked. The main benefit of such a *call-site-specific specialization* approach is that for cases in which an effective global policy cannot be derived for a given function, at least the majority of its call sites can still be protected. For instance, when just a few call sites of `VirtualProtect()` legitimately use the execute permission in a given application, then an attacker would have to invoke it only through those particular call sites, as part of a ROP payload that needs to allocate executable memory.

In a sense, this approach creates *multiple versions* of the original function, each with different stripped functionality. The aggressively stripped-down version becomes the default (as it is of no use for attackers), while invocations to the more "dangerous" version of the function can be permitted only from the specific code locations that really require the sensitive functionality (i.e., allocating executable memory). To prevent attackers from simply jumping to particular call sites and reusing the sensitive version of the function, the invocation points can be protected further by applying context-sensitive and type-based control flow integrity [14]. Assuming such a form of CFI is in place, this enforcement complicates significantly the construction of a functional ROP payload, as the attacker has now less freedom in picking appropriate ROP gadgets that include

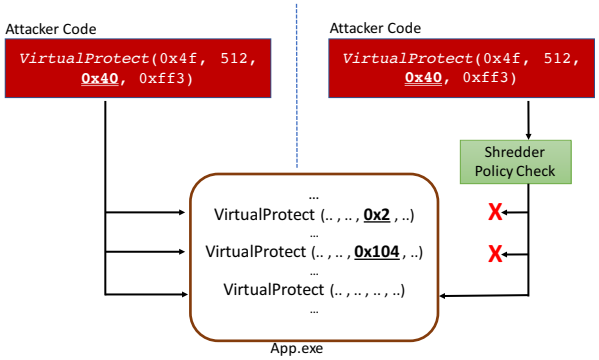

Figure 7: Call-site-specific policies restrict the call sites that can be reused by an attacker to a fraction of all available call sites. In this example, when Shredder is enabled, only the third call site remains usable.

the non-protected call sites, while at the same time not violating the CFI policy [12, 15, 26, 59, 61].

Figure 7 illustrates this case further. The attacker's code attemtps to use `VirtualProtect()` with the third parameter set to `0x40`. The application in this case has three different call sites for that function, two of which has known, hard-coded values (`0x2` and `0x104`), which now become part of the call-site-specific policy. Given that these known values are different than what the attacker needs to use for the exploit code, if any of the first two call sites is reused as part of a ROP payload, then Shredder will identify the argument mismatch and prevent the attack. Only if the last call site is used by the attacker, then the attack will succeed.

Call-site-specific policies are implemented by including call site identifiers to the policies. Every call site of a critical function is recorded along with the set of input arguments. The address of each call site is kept with respect to the subroutine it is part of and with respect to the base of the binary image. Also, the name of the module (executable or shared library) which makes the call is also recorded as part of the policy.

A policy for a given call site includes the following fields:

(1) Critical function name
(2) Module name
(3) Offset within the module
(4) Tuple with "known" and "unknown" arguments for this particular call site

For every call of a sensitive function, the runtime interposition layer traverses the stack of the process to determine the origin of the call. For the current process, we use `EnumProcessModule()`, `GetModuleFileName()` and `GetModuleInformation()` to get the image base addresses of all the modules loaded by the application. With this information, an intercepted call is verified against the policies of all modules.

Table 4 shows the number of call sites for which Shredder was previously unable to create global policies (second column), and their subset for which site-specific policies can be derived. In several applications, we see that a majority of the previously unprotected call sites are now protected. Still, there are cases like 7zip and

Table 4: Number of call sites for which site-specific policies can be derived.

Application	Call sites without global policies	Call sites with site-specific policies
7Zip	12	0 (0%)
Google Chrome	42	31 (75%)
Microsoft Edge	94	52 (55%)
Mozilla Firefox	28	12 (43%)
iTunes	108	76 (70%)
PhotoViewer	4	2 (50%)
Notepad++	18	0 (0%)
Powershell	2	0 (0%)
VLC	5	2 (40%)
Winrar	47	11 (23%)

Notepad++ for which even site-specific policies could not be derived. As discussed in Section 5.3.2, this is again mostly due to input arguments that can only be derived at runtime.

B ROP PAYLOAD DATA SET

Table 6 shows a detailed list of the ROP payloads used in our evaluation (some sources contain multiple distinct payloads). We did not include payloads that would not be functional in the considered Windows 10 environment (e.g., ROP payloads that rely on `setProcessDEPPolicy()`, which works only on Windows XP). The vast majority of the payloads use either `VirtualProtect()` to give execute permission to the memory area where the second-stage shellcode resides, or `VirtualAlloc()` to allocate some executable memory and copy the shellcode there. To achieve this, there are specific sets of values for certain arguments that exploit code must use. For instance, the `flNewProtect` or `flProtect` argument, respectively, should be one of `PAGE_EXECUTE`, `PAGE_EXECUTE_READ`, `PAGE_EXECUTE_READWRITE` (most commonly encountered), or `PAGE_EXECUTE_WRITECOPY`. In applications that these memory protection constants are never used, Shredder is able to derive policies that effectively prevent their use. We also encountered a few payloads that use `WinExec()` or `NtSetInformationProcess()`.

Table 5: Set of critical Windows API functions considered by Shredder for policy enforcement.

kernel32.dll	
CloseHandle	GetSystemDirectoryA
CreateFileA	GetSystemDirectoryW
CreateFileMappingA	GetTemplatePathA
CreateFileMappingW	GetTemplatePathW
CreateFileW	LoadLibraryA
CreateProcessA	LoadLibraryW
CreateProcessW	PeekNamedPipe
CreateRemoteThread	ReadFile
DeleteFileA	Sleep
DeleteFileW	VirtualAlloc
DuplicateHandle	VirtualProtect
ExitProcess	WaitForSingleObject
ExitThread	WinExec
GetCurrentProcess	WriteFile
ws2_32.dll	
accept	recv
bind	send
closesocket	socket
connect	WSASocketA
ioctlsocket	WSASocketW
listen	WSAStartup
wininet.dll	
InternetOpenA	InternetOpenW
InternetOpenUrlA	InternetReadFile
InternetOpenUrlW	
msvcrt.dll	
_execv	fopen
fclose	fwrite
urlmon.dll	
URLDownloadToFileA	URLDownloadToFileW
ntdll.dll	
NtSetInformationProcess	

Table 6: List of Windows ROP payloads used in our experimental evaluation.

1–5)	Direct RET: The ROP Version with Immunity Debugger, Direct RET: Generic parameter generation for ROP Direct RET: NtSetInformationProcess(), Direct RET: WinExec(), Direct RET: Using VirtualAlloc() https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/
6)	ASLR/DEP bypass exploit - BlazeDVD5.1 https://thesprawl.org/research/corelan-tutorial-10-exercise-solution/
7)	A DEP evasion technique http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html
8)	Buffer overflow attacks bypassing DEP http://www.mastropaolo.com/2005/06/05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/
9)	TrailOfBits Practical ROP https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf
10)	The Audio Converter Case http://tekwizz123.blogspot.com/2014/02/bypassing-aslr-and-dep-on-windows-7.html
11)	DEP Bypass https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2016/june/writing-exploits-for-win32-systems-from-scratch/
12)	Defeating DEP with ROP https://samsclass.info/127/proj/rop.htm
13)	Vulnserver DEP Bypass Exploit https://web.archive.org/web/20121110045053/http://www.violentpython.org/wordpress/
14)	Malicious PDF in Adobe Reader http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0569
15)	Malicious SWF in Adobe Flash https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2883
16)	Abusing Non-ASLR'd Modules https://exploitresearch.wordpress.com/2012/06/23/abusing-non-aslrd-modules-on-windows-7/
17)	Buffer Overflow on Vulnserver http://resources.infosecinstitute.com/return-oriented-programming-rop-attacks/
18–19)	DEP bypass with msvcr71, mona.py, DEPS-Precise Heap Spray - Firefox, IE10 https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/
20)	Bypassing Win ASLR using "skype4COM" http://www.greyhathacker.net/?p=641
21)	Whitepaper on Bypassing ASLR/DEP https://www.exploit-db.com/docs/17914.pdf
22)	An Easy Guide to Bypass DEP using ROP http://securitydynamics.blogspot.com/2015/11/an-easy-guide-to-bypass-dep-using-rop.html
23)	A-PDF All to MP3 Converter http://www.exploit-db.com/exploits/17275/
24)	Integard Pro v2.2.0.9026 http://www.exploit-db.com/exploits/15016/
25)	Mplayer Lite r33064 http://www.exploit-db.com/exploits/17124//
26)	Adobe Acrobat Bundled Int Overflow http://www.exploit-db.com/exploits/16670/
27)	Adobe Flash "newfunction" Invalid Ptr http://www.exploit-db.com/exploits/16687/
28)	Adobe CoolType SING "uniqueName" http://www.exploit-db.com/exploits/16619/
29)	Adobe Flash "Button" Remote Code Exec http://www.exploit-db.com/exploits/16667/
30)	Winamp v5.572 http://www.exploit-db.com/exploits/14068/

Table 7: Sets of equivalent Windows API functions considered for the construction of shellcode variants with the same functionality (used in the experiments of Section 5.4.3).

1)	VirtualAlloc, coTaskmemAlloc, globalAlloc, heapAlloc, localAlloc, malloc, new
2)	VirtualProtect
3)	CreateThread, CreateRemoteThread, CreateProcess, ShellExecute, ShellExecuteEx, system, WinExec
4)	CloseHandle, FindClose
5)	CreateFile, CreateTextFile, WriteAllText, WriteAllLines, Write, WriteLine, WriteAsync, WriteTextAsync, WriteLinesAsync, WriteLineAsync, AppendLinesAsync, AppendTextAsync, AppendAllLines, AppendAllText, AppendText, WriteFile, FtpGetFile, FtpPutFile, FileOpenPicker
6)	OpenFile, FtpOpenFile, ReadFile, CreateFileMapping
7)	DeleteFile, remove, unlink, MoveFileEx, MoveFileTransacted, MoveFileWithProgress, FtpDeleteFile, _wremove
8)	DuplicateHandle, CreateFile, WSADuplicateHandle
9)	ExitProcess, ExitThread, TerminateProcess, _tsystem
10)	closesocket, shutdown
11)	ioctlsocket, WSAAsyncSelect, WSAEventSelect, WSAIoctl, recv with MSG_PEEK
12)	URLDownloadToFile, ShellExecute, libcurl, InternetReadFile, InternetOpenUrl
13)	InternetOpen, open, WinInet, HttpOpenRequest, HttpSendRequest, InternetOpenUrl, FtpGetFile, FtpGetFileEx, FtpOpenFile
14)	send, HttpSendRequest