On the Effectiveness of Code Injection Mitigation Technology to Increase Exploit Development Costs Significantly

Adrián Barreal
abarreal@fundacionsadosky.org.ar

STIC, Fundación Sadosky, Argentina

Abstract

In an effort to increase exploit development costs, Microsoft released several mitigations that attempt to prevent attackers from achieving arbitrary code execution in the context of the protected process. What these mitigations currently prevent, however, is the execution of injected code, still allowing attackers to construct arbitrarily complex payloads by means of code reuse techniques. While current technique tends to make pure code reuse payloads difficult to write and also target specific, in this paper we will see how attackers may adopt a Remote Procedure Call based exploitation model to easily construct complex, modular, maintainable, and target independent payloads with minimal code reuse, hence keeping their exploit development costs relatively low even when lacking code injection capabilities.

1 Introduction

Despite the numerous mitigation technologies that have been developed and deployed over the years, and despite significant efforts by the industry to reduce bug density in software, memory corruption vulnerabilities are still being found and exploited in the wild [1][2]. Coming to terms with this idea, Microsoft has recently moved from a preventative software security approach, in which the focus is on preventing bugs, to a proactive, more general approach in which the main strategy is to make difficult and costly for attackers to find, exploit and leverage software vulnerabilities [3][4]. Born from this effort were a set of attack mitigation technologies that aim to deprive attackers from the highly desirable capability of arbitrary code execution. Attackers with arbitrary code execution capabilities will often proceed to have the compromised process execute their "payload," a piece of code that will, in the context of the vulnerable process, perform arbitrary tasks beneficial to the attacker (exfiltrating sensitive data or installing malware, for example) [2][4]. Code execution capabilities come essentially in two flavors: code injection and code reuse. In the former, payloads are written in high level code, compiled to machine language, and then injected into the target process to be executed; in the latter, the attacker is limited to reusing code that already exists in the memory space of the vulnerable application. Payloads based on the former may be modular, easy to write, easy to maintain, and portable across different targets; payloads based on the latter are based in techniques such as Return Oriented Programming (ROP) [10], and tend to be monolithic, difficult to write, difficult to maintain, and often target specific. Sensibly, if the goal is to make exploit development more expensive, neutralizing code injection and leaving attackers only with code reuse would seem to be an effective tactic. Current mitigations deployed in Windows 10 that attempt to eliminate code injection based exploits are Data Execution Prevention (DEP), Arbitrary Code Guard (ACG), Code Integrity Guard (CIG) and Child Process Policy (CPP) [4][5]. As far as is known today, the combination of these technologies prevents attackers from achieving arbitrary code execution by executing injected code. In this work, however, we set out to show how, even without code injection capabilities, attackers may still be able to develop arbitrarily complex, modular, maintainable, and target agnostic payloads in high level code, based on minimal code reuse sequences, by adopting an exploit architecture based on the Remote Procedure Call model of computation [8]. In addition, we analyze how attackers may adapt their overall exploitation strategy to deal with the lack of code injection capabilities.

2 Related Work

In [6], Fratric set out to answer the question of how useful ACG and out-of-process JIT [4] would be in preventing an attacker from exploiting Microsoft Edge. In his work, he states how the lack of executable code injection capabilities may affect exploit writers' workflow. The most likely alternative strategies enumerated by Fratric that attackers may now adopt are essentially the following:

1. Attackers may choose to execute a data-only attack, which would allow them to gain further privileges. In the context of browser exploitation, for example, it may allow them to weaken Same Origin Policy, which would allow for other kinds of attacks.

2. In scenarios where a scripting engine is not present, the only known avenue left to attackers is to develop their entire payload as ROP. With current technology, this would be a time-consuming process; however, technology could evolve to make ROP-only payloads easier to write.

3. If a scripting environment is present, attackers with read/write primitives could leverage said scripting environment to create an interface that would allow them to call arbitrary native-code functions from the scripting engine while providing arbitrary arguments, and then read back the returned value to continue executing their payload in high level code.

Fratric states, "it may seem that ACG is not going to be very useful, especially in a web browser." However, a web browser context may not be a requirement for code injection mitigations in general to fail to prevent attacks. In this work we focus on items 2 and 3. Concretely, we will see that case 3 is not limited to browsers or scriptable engines only; even in case 2, when a scripting engine is not present (e.g. in remote server exploitation scenarios), by adopting an exploit architecture as described in section 5, attackers may still be able to develop an interface as described in list item 3, which would allow them to write their payload mostly in high level code, keeping it modular, maintainable, and target independent, while keeping dependencies on code reuse techniques minimal.

The exploitation mechanisms described in this work are in spirit similar to the RPC syscall proxying method detailed by Caceres in [9]. However, to deal with code injection mitigations, the syscall server cannot be injectable but must be made transient and code reuse based instead. How to achieve this will be covered in detail in section 5.

3 Contributions

This paper's contributions are the following:

- A new exploitation technique, the Remote Procedure Code Reuse Attack (section 5), and an analysis of an exploit architecture based on it that would allow attackers to create complex, modular, maintainable, target independent remote exploitation payloads in high level code, with minimal dependency on code reuse techniques. In addition, a proof of concept written in C that exemplifies such an architecture is provided[1].
- A generalization of the techniques described by Fratric in [6] with the goal of estimating what avenues code injection restricted attackers may be willing to explore in case that code injection mitigations become more widely adopted (section 6).
- An analysis that attempts to answer the question of whether current mitigations that attempt to prevent code injection may be effective at increasing exploit development costs significantly and permanently. Along the report, a series of observations will be made that will hopefully allow us to conclude with a convincing answer in section 7.

4 Technical Background

Some software bugs cause the corruption of the memory of the process that instantiates the program. These memory corruptions may be abused by an attacker to achieve control over the memory space of the vulnerable target process. Such control may scale up to arbitrary read and write capabiliites; that is, an attacker may be able to read arbitrary memory content or write arbitrary data from outside the process. Additionally, by taking over the instruction pointer, the attacker may be able to divert control flow, hence taking control of the program execution and potentially executing arbitrary code. Originally, exploiting such vulnerabilities to achieve arbitrary code execution was, in today's standards, a relatively trivial task: attackers could simply write machine code (the shellcode) somewhere in the memory of the target process, and then divert control flow toward the injected code to execute their arbitrary payload [17]. Since then, operating systems have been fortified with several attack mitigation technologies; we now proceed to enumerate those that are most relevant for this work:

Runtime Checks (e.g. stack cookies): These provide a basic mechanism to detect memory corruption at runtime. Stack cookies (or stack canaries), for example, are values pushed into the stack at the beginning of each protected function f such that, when f is about to return, if the cookie in the stack differs from the "real" value of the cookie, stored somewhere outside the stack, then some buffer must have been overflown and execution must be aborted. Attackers may deal with stack cookies by abusing write vulnerabilities that skip over the cookie in the stack; alternatively, vulnerabilities that allow exfiltrating memory (information leaks [14]) may allow the attacker to read the real value of the cookie before corruption, hence being able to preserve it after the eventual overflow.

Write XOR Execute (W $\oplus$ X) / *Data Execution Prevention* (DEP): To make exploitation more difficult, mitigations based on page permissions were implemented. The idea of

[1] The proof of concept will be made available in `https://github.com/programa-stic/rpcras-poc`.

W⊕X (or DEP, in Windows) is to make memory pages either writable or executable, but not both. This would prevent attackers from injecting shellcode that they could execute. Modern exploits bypass this mitigation by applying a technique known as Return Oriented Programming (ROP) [10]. The idea of ROP is to chain small code snippets called gadgets, pre-existing in the target process' address space, to build complete payloads. Several mitigations attempted to neutralize ROP, but they turned out to be either impractical or ineffective [15][16]. In any case, the proposal of these mitigations also gave rise to additional exploitation techniques, also based on code reuse, such as Jump Oriented Programming (JOP) [11] and Counterfeit Object Oriented Programming (COOP) [12].

Address Space Layout Randomization (ASLR) [13]: To divert execution flow to the injected shellcode, or to any useful gadget, the attacker must be able to locate them in the address space of the target process. ASLR attempts to prevent an attacker from locating specific memory objects by randomizing their locations. To defeat ASLR, attackers typically resort to information leak vulnerabilities that allow them to map the address space of the target process.

Arbitrary Code Guard (ACG): While potentially very powerful, ROP chains up until recently could be tipically very simple: they would often call a memory management function such as `VirtualProtect` in Windows or `mprotect` in Linux to make injected shellcode executable, hence disabling W⊕X / DEP and transitioning quickly into a shellcode based payload. To prevent this, Microsoft released ACG, a mitigation technology that, when enabled for some given process P, guarantees that new executable memory in P cannot be allocated, and existing executable memory in P cannot be modified [6][7].

Code Integrity Guard (CIG): If writing shellcode directly is not an option, attackers may choose to load a library instead. To combat this strategy, Microsoft implemented CIG, a mitigation that complements ACG by preventing the protected process from loading unsigned code into its memory space.

Child Process Policy (CPP): In addition to ACG and CIG, Microsoft also implemented CPP, which prevents the protected process from instantiating children. This prevents attacks that rely on spawning new processes with weakened security policies [4].

These mitigations enabled together should prevent attackers from injecting executable code into the target system, limiting them to code reuse attacks only. For this work, we will be effectively assuming such restriction.

5 Remote Procedure Code Reuse Attacks

In this section we introduce the concept of the Remote Procedure Code Reuse Attack (RPCRA). We begin first by proposing a threat model in section 5.1. Then, we introduce the exploitation model and a concrete case in section 5.2. Finally, in section 5.3, we analyze an exploit architecture based on RPCRAs that decouples the payload from the target, using the proof of concept as reference. How these techniques may be incorporated into an overall exploitation strategy is studied in section 6.

5.1 Threat Model

Suppose that we have an attacker A and a target process P_0. Let us assume P_0 to be a persistent process such that A can repeatedly interact with P_0 through any sort of message exchange mechanism and, under normal operation, P_0 remains indefinitely available to serve further client requests. Let us also assume P_0 to be protected by a set of mitigations that prevent code injection based attacks. Suppose then that attacker A is aware of a set of memory corruption vulnerabilities in P_0 that would allow him/her to achieve the following capabilities in the context of the process:

- Read and write in some mapped (i.e. addresses known) region of the memory space of the process (restricted by page permissions). Notice that this is less restrictive than arbitrary read and write; linear overreads and overflows may still provide for this capability.

- Code execution (only through code reuse, as code injection is prohibited).

Now, attacker A would like to execute a complex payload in the context of P_0. Since the target is immune against code injection, A is restricted to code reuse techniques only. While we may argue that A cannot execute *arbitrary code* in the context of P_0, notice that this does not prevent A from constructing an *arbitrary payload* from non-arbitrary code (i.e. already existing code); while A cannot inject and execute arbitrary machine code into P_0, it has been shown that, in realistic attack scenarios, many code reuse techniques are Turing Complete [10][11][12][2, p. 53]. Therefore, we may make the following observation:

Observation #1*: Mitigations that successfully prevent attacks based on code injection do not necessarily prevent attackers from executing arbitrary payloads in the context of a compromised process.*

A positive remark is that, if A already had a set of shellcode payloads that they relied on for serial exploitation, the lack of code injection capabilities may indeed impact A's exploitation process. As we mentioned, however, this does not prevent A from developing new code reuse based payloads to further fulfill his/her exploitation needs. It is true, however, that with current technology, arbitrary code reuse payloads tend to be difficult to write and also target specific. If the goal is to disrupt the economy of exploits, cannot we expect that imposing this difficulty on attackers should be a big step in the right direction? The answer could only be affirmative if such difficulty is not fleeting but remains constant in time. We may conjecture, however, that writing payloads based on code reuse only is difficult today because, up until now, there has not been any real incentive to develop automated tools or alternative exploitation models to facilitate the task. We continue, then, by proposing such a model, and analyzing a concrete exploitation scenario in which it may be applied.

5.2 Exploitation Model

As code reuse payloads need to increase in complexity to deal with ever evolving attack mitigation technology, they will also need to become more flexible and manageable. One first step toward that goal may be that of modularization. Let us now define the concept of *Code Reuse Procedure* (CRP). Given target process P_0, we may define a CRP r to be a functional unit that consists of some static data d and a small code reuse payload p that depends on d. To execute r in the context of P_0, attacker A must inject the unit (both payload and data) into the memory space of P_0 (through write primitives) and then get the process to execute the payload (through control flow hijacking). Once execution starts, payload p would perform some short procedure that depends on d, write some output data to a location later readable by the attacker (through read primitives), and then return execution flow back to normal for continued interaction. If A can find a mechanism that would allow him/her to sequentially inject CRPs into P_0's address space, get them executed, and read their output, A could implement what we may call a *Remote Procedure Code Reuse Attack* (RPCRA), inspired by the Remote Procedure Call model of computation. In said attack, the attacker would implement complex operations in high level code outside the target, requiring code reuse only to call specific API functions in the vulnerable process. The overall procedure is schematized in figure 1.

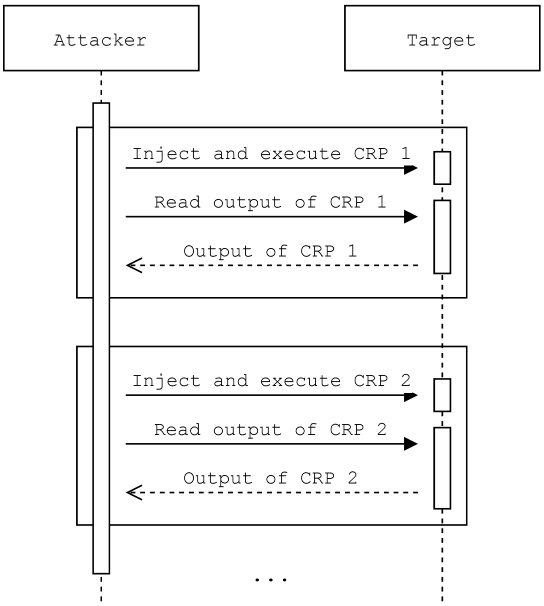

Figure 1: *Diagram of a remote procedure code reuse attack. The attacker sequentially injects code reuse procedures into the target process and reads their output.*

For a concrete example, let us analyze the classic stack buffer overflow case. Suppose that we have a buffer overflow/overread scenario. In this case, an attacker may first exploit the overread vulnerability to read the contents of the stack, potentially leaking canaries and pointers, hence defeating ASLR and stack protection. Then, the attacker may inject an overly long byte sequence to overflow the stack buffer, gaining control of the function's frame return address and achieving code reuse through ROP. Suppose then that, before corruption, the stack frame of the vulnerable function `f` is as shown in figure 2.

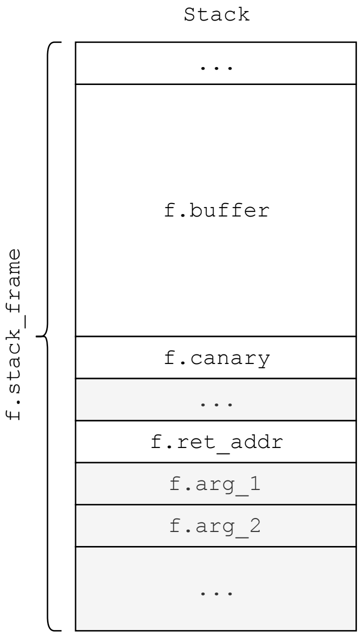

Figure 2: *Vulnerable function's stack frame before corruption.*

In this context, the attacker can read and write anything from `f.buffer` in the direction of `f.canary` and `f.ret_addr`. Now suppose that, after having performed read procedures to read target's stack content, the attacker abuses the buffer overflow write primitive to induce a scenario as depicted in figure 3. In this example, the attacker has injected, starting at the frame's return address, a *pro-*

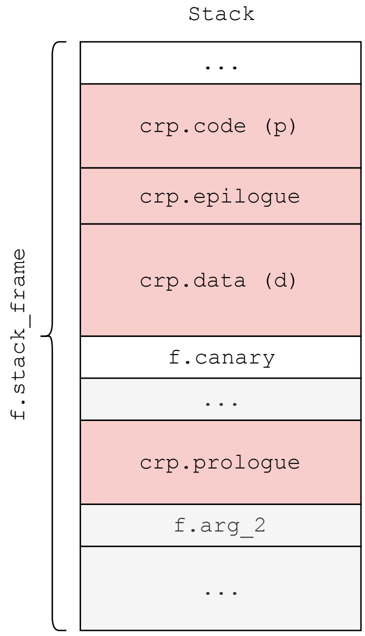

Figure 3: *Vulnerable function's stack frame after corruption. A modular CRP has been injected.*

logue for the CRP, a small ROP chain that will be first to execute when the function returns. This prologue may be a small stub injected to bootstrap the main CRP's payload, what has been labeled as `crp.payload` (p) in figure 3. For instance, in the proof of concept (to be further detailed in section 5.3), the prologue is a 16 byte stack pivot that moves the stack pointer to the address where payload p starts; notice that what the attacker is essentially doing is creating a stack frame for his/her CRP. A priori, there is no length limitation imposed on the prologue; however, care must be taken to preserve as much state as required to resume proper execution when the code reuse procedure finalizes.

Once the CRP has been setup, payload chain p executes a short sequence of instructions, likely dependant on data d, and then possibly writes some output data to a location later readable by the attacker by means of read primitives. For the case of the proof of concept, output data is written to the first few bytes after the last return address near the bottom of the stack, where it may not be accidentaly overwritten nor it may affect program's proper execution flow. Depending on the available exploitation primitives, however, output (and input) data locations may vary.

Then, after the payload has been executed, an *epilogue* restores execution flow to its former state, resuming program execution (ideally) as if nothing had happened. For instance, the epilogue for the proof of concept first restores the original return address of the vulnerable function, and then performs a second stack pivot to restore the original stack pointer. As a side note, depending on the specific target, the epilogue may actually have some side effects; in any case, as long as these side effects still allow for continued CRP injection, from the attacker's perspective this may not be an issue.

Finally, once program's proper execution flow has been restored, the attacker employs a read primitive to retrieve the output data "returned" by the CRP. The attacker may then handle said data in high level code, possibly preparing additional CRPs to be injected into the target process. With this powerful primitive, an attacker may be able to build a complex payload in high level code, only using code reuse to execute specific API calls in the target by means of CRP injection. Some ways in which attackers may make use of this mechanism are explored in section 6.

5.3 Exploit Architecture

In this section we analyze an exploit architecture based on the remote procedure code reuse model that allows for target independent payloads to be written in high level code. For reference, we will be using the proof of concept exploit; the PoC was designed to exploit a simple Windows 10 based echo server that presents stack buffer overflow/overread vulnerabilities, showcasing an RPCRA just as described in section 5.2. The payload executes a sequence of CRPs to copy several data structures from the attacker's machine to the vulnerable process' memory space, and performs required calls to register a task with the remote task scheduler. The architecture of the exploit is as schematized in figure 4. The architecture follows a layered design in which each layer consumes the services provided by the adjacent below layer.

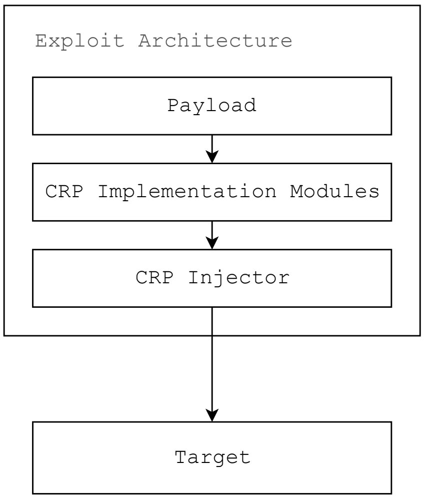

Figure 4: *High level overview of the Proof of Concept exploit. Each layer consumes the services exposed by the layer immediately below.*

Notice that the payload is actually at the top of the structure, and it depends only on procedures exposed by the CRP implementation modules. CRP implementation modules orchestrate usage of the basic primitives exposed by the injector to develop more complex functionality. Both payload and CRP implementations are written exclusively in high level code. At the bottom of the structure, on the other hand, we have the injector. The injector is the one that constructs CRP functional units (including code, data, prologue and epilogue), injects them into the target process to be executed, and retrieves output data. The injector for the PoC has been designed to expose a generic interface such that the only target dependant component of the system is the injector itself. That is, the only component that knows the details of the exploitation procedure is the injector; other components are target agnostic and they trust the injector to get their procedures executed in the target process for them[2]. This allows for payloads and CRP implementations to be target independent as long as a proper injector that exposes the specified interface can be developed. To provide for target independent payloads, the injector must expose at least the following functionality:

Generic remote function calls or system calls: The proof of concept injector exposes a function `remote_find_and_call`, which has the following signature:

```
int32_t remote_find_and_call(
    char *lib_name,
    char *proc_name,
    uint8_t *data,
    uint64_t data_len,
    uint8_t **out,
    uint8_t arg_count,
    ...);
```

On call, the injector will first execute CRPs to load the library with name given by `lib_name` into the target process' memory space; then it will find, in that same library, the procedure with name given by `proc_name`. To avoid redundant calls, the PoC injector implements a local caching system for library handles and procedure addresses. Argument `data_len` tells the injector to embed `data_len` bytes from `data` into the CRP unit as static data d to be used by the remote procedure. If the CRP is expected to return some value, pointer `out` may be provided by the caller. If

[2] That being said, this does have its limitations, as CRP definitions may still be architecture and operating system dependent.

out is not null, the injector will, before returning, use read primitives to copy output data in the remote output storage section (i.e. where payload p writes output to) to a local buffer, which *out will be made to point to. The caller may then read output data by dereferencing this pointer. Finally, the variable length arguments are those that will be passed to the remote function when being called in the context of the target; arg_count arguments will be expected.

The actual exploitation procedures and code reuse sequences required to bootstrap the CRP are handled internally by the injector. An interesting thing to notice is that the PoC injector makes use of only fourteen ROP gadgets in total, including those for LoadLibrary, GetProcAddress, prologue and epilogue, and they all were defined in a single place; everything else was built in C code, reusing those fourteen gadgets over and over in consecutive CRP calls. Concretely, to handle any call, the injector internally defines what essentially is the following ROP chain:

```
uint64_t generic_crp_code[] = {
    LOAD_RCX(arg_1),
    LOAD_RDX(arg_2),
    LOAD__R8(arg_3),
    LOAD__R9(arg_4),
    target_function_address,
    push_rsp_gadget,
    0,
    0,
    0,
    0,
    arg_5,
    arg_6,
    /*
     * Additional arguments. Can be
     * a fixed amount, enough to handle
     * most functions; unused arguments
     * can just be 0. This also allows
     * for a fixed push_rsp_gadget.
     *
     */
    arg_n,
    SAVE_RAX_IN_OUTPUT_STORAGE
}
```

As we can see, it's just a plain function call with a variable number of arguments: the payload will first load parameters 1 to 4 into registers rcx, rdx, r8 and r9, respectively; additional arguments are provided after the return address. Then, after execution, the target procedure will return to a gadget that will have the stack pointer skip over the static argument vector, arriving finally to the last gadget that will write the value returned in rax to the first qword of the output data storage. What follows afterwards is an epilogue that will return execution flow back to normal for continued interaction. Since the remote procedure model allows for complex logic to be handled client side, only CRP prologue, function call and CRP epilogue need to be implemented through code reuse; the prologue and epilogue of the PoC injector are also short, with one and three gadgets, respectively. This may imply that code reuse attacks could not only be powerful, but they could also be modular, easy to write and easy to maintain. Even further, if we consider that the above exposed ROP chain is actually very simple, it may even be possible to generate it automatically. Given that this construct may allow for payloads of arbitrary complexity, we may make the follwing observation:

Observation #2: *Payloads of arbitrary complexity may still be built from short, simple code reuse chains. These chains may be short enough to be easily maintainable, and possibly simple enough to be automatically generated.*

Abstract data references: To decouple the payload from any target, the injector must provide abstract addressing mechanisms. That is, the payload must not know beforehand any address in the target's address space, but it must be still able to find the address of the output storage (to provide out pointers to output storage as arguments) and addresses of input data in the static data section (to provide input pointers to static data as arguments). To find the former, the PoC injector exposes a function `get_out_storage_base` that returns base address the output storage section. For the later, the injector exposes a rudimentary labelling mechanism with precompiler macros: before injection, the injector will iterate through the CRP buffer replacing `MAKE_REFERENCE(id)` labels for the address of the corresponding `MAKE_TARGET(id)`, plus 8 bytes. This allows for remote procedure calls that look as follows:

```
/* Create a static data buffer to inject
 * with the CRP payload. The first qword
 * will be replaced by a MAKE_TARGET
 * label for the string address to
 * be used as an argument.
 *
 */
uint8_t static_data[] =
    "\x00\x00\x00\x00\x00\x00\x00\x00"
    "86d35949-83c9-4044-b424-db363231"
    "fd0c@ncalrpc:";

*(uint64_t*)static_data = MAKE_TARGET(1);

/* Perform remote call. Notice the usage
 * of MAKE_REFERENCE and get_out_storage_base
 * to leave the task of locating input
 * and output addresses to the injector.
 *
 */
remote_find_and_call(
    "rpcrt4.dll",
    "RpcBindingFromStringBindingA",
    (uint8_t*)static_data,
    sizeof(static_data),
    &out,
    2, // argc
    MAKE_REFERENCE(1),
    get_out_storage_base() + 8);
```

This sequence executes the remote procedure `RpcBindingFromStringBindingA` in the target, passing as the first argument the remote address of the binding string injected with the CRP as static data, and passing as the second argument the address `get_out_storage_base() + 8` to write output binding handle to. Then, after the call returns and pointer `out` has been set, output may be read in the client as follows:

```
uint64_t ret_val = *(uint64_t*)(out + 0); //
↪  Read returned value (rax).
uint64_t out_hnd = *(uint64_t*)(out + 8); //
↪  Read output handle.
```

Size restrictions: Some CRP implementations may require at least a minimum amount of space available for their static data, and the injector may expose functionality to query for such restrictions. Knowledge of these limitations may allow for adaptable CRPs; the CRP implementation library of the PoC, for example, exposes a function `copy_to_target` that copies large local data buffers to the remote heap in chunks of maximum size determined by `get_max_input_data_size()`, a function exposed by the injector. Other CRPs in the PoC use `copy_to_target` to copy several large data structures and strings to the memory space of the vulnerable process. Notice that this highlights an advantage that RPCRAs have over traditional monolithic ROP chains: since data can be copied in chunks, they may not be as space constrained as traditional payloads; while the buffer in the PoC target is only 512 bytes long, several kilobytes of data are copied to the server's heap during payload execution. If an equivalent payload was to be implemented as a monolithic ROP program, the available space would be barely large enough to hold the ROP chain and some static strings, but not all of the data structures required to conclude the final call to the task scheduler, even if frames of other functions were also overwritten.

Provided an injector with these characteristics, CRP implementation modules may be built on top to expose an interface to payload writers. The PoC exploit implements several basic functions such as a remote `malloc` and a remote `memset`, and then there is also a remote task scheduler module that performs local RPC calls from the target process to register a task defined by the caller.

6 Threat Analysis

In this section we study how restrictions on code injection may affect the attacker's exploitation procedures and strategies to compromise a system composed of several interacting processes. Suppose that we have a scenario as depicted in figure 5. In this scheme, we have an attacker A and a system S, which includes at least a network facing process $P_0 \in S$, protected by a set of mitigations that prevent code injection based attacks. There are, in addition, internal processes $P_1, P_2, \ldots, P_n \in S_L \subset S$, each of which expose an internally available Inter-Process Communication (IPC) interface accessible from P_0. Internal processes may or may not be as well protected as P_0, and among them, there may or may not be one that is susceptible to code injection attacks. In addition, each process $P_j \in S$, $j = 0, 1, \ldots, n$, may have a different set of abstract privileges V_j. Furthermore, all processes in S rely on services provided by an underlying system K such that controlling K would grant total control over S (i.e., K has at least all privileges possesed by all processes in S).

Suppose then that, among the whole set of privileges $V = \cup_{j=0}^{n} V_j$, there is a privilege $v \in V$ that attacker A wants to obtain. In this section, we analyze the strategies that attacker A may adopt to obtain privilege v in system S, and the role that code injection mitigations may play in preventing such compromise.

Initial exploitation: Suppose that A knows of one or more vulnerabilities in P_0 that would allow him/her to achieve code execution capabilities in the context of P_0. Again, since code injection is restricted, A is limited to code reuse payloads only. By applying the techniques that we studied in sections 5.2 and 5.3, however, A may still be able to develop an arbitrarily complex payload. At this instance, if $v \in V_0$ and there are no additional mitigations, then attacker A has already achieved his/her purpose, and this is the trivial case which we have already seen. But let us analyze some additional cases to see what else an attacker may do.

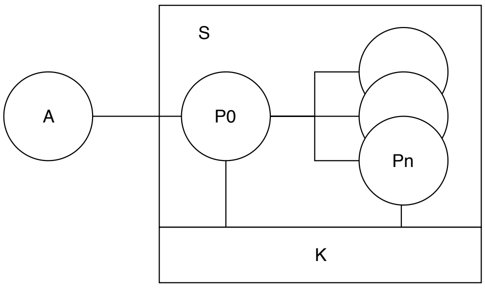

Figure 5: *Attack scenario. Attacker A may interact with the network facing process P_0 at the border of system S. Inside S, P_0 may interact with several other processes $P_1, P_2, \ldots, P_n$ through IPC mechanisms. A privileged system K underlies S, and its interface is accessible from any process $p \in S$.*

Attack proxy: Given that, by performing an RPCRA, A may be able to execute arbitrarily complex payloads in the context of P_0 without crashing the process, leaving it available for continuous service, it may be possible for the attacker to use P_0 as a window into the system without ever writing a single file nor launching any additional listener. What A can do, therefore, will be only limited by the privileges of P_0. We may then make the following observation:

Observation #3: *Where RPCRAs are possible, enforcing least privilege and monitoring network and process activity becomes* *critical to prevent "invisible" attackers from using border processes as windows into the system.*

Inter Process Code Reuse and Exploitation, Privilege Escalation: Suppose now that A has a really sophisticated, injectable payload that they would like to execute in S, and rewriting it through RPCRAs is not an option. Code injection mitigations prevent A from executing his/her payload in P_0 but, since payload complexity is a priori not limited, nothing prevents A from achieving code execution in the context of some other process $P_j \in S_L$ which P_0 may interact with through any exposed IPC interface. The most trivial case would be that of a "confused deputy" [18] which allows for local processes to get code executed for them; for example, the proof of concept exploit shows how, if P_0 is not sandboxed and its permissions are not properly restricted, an attacker may register a task with the task scheduler to launch any arbitrary executable. In general, if there is some $P_j \in S_L$ such that $v \in V_j$, then A may look for ways to make P_j use privilege v in his/her favor. If a confused deputy is not immediately available, the attacker may resort to plain exploitation to escalate privileges. If there is some process $P_j \in S_L$ such that $v \in V_j$, and P_j is not protected against code injection, then A may try to exploit P_j from P_0 to achieve arbitrary code execution. While this may be unfeasible with monolithic code reuse payloads, RPCRAs could make it possible. Local privilege escalation exploits may even be adapted to fit an injector, allowing for remote exploitation with minimal changes. Alternatively, A may choose to go for K to achieve complete system compromise. Notice that, again, enforcing isolation and least privileges becomes critical to prevent attackers from propagating into the system. We may then make the following observations:

Observation #4: *Where RPCRAs are possible, enforcing least privilege and monitoring network and process activity becomes critical to prevent attackers from escalating privileges and propagating into the system.*

Observation #5: *If there is any process $P \in S_L$ that is not fully protected by code injection mitigations, attackers may still attempt to exploit P from P_0 to get their pre-existing injectable payloads executed.*

We may argue that the methods proposed by Fratric in [6] to defeat ACG in Edge are roughly of this category: while JIT compilation has been moved out, the JIT server is still accessible from the content process; hence, attackers can still trick the JIT server through different means to get their payload executed.

Finally, we may conclude this section by stating that, in order to prevent future system compromise, more effort should be made to improve mitigations that attempt to prevent control flow hijacking in the first place, such as CFI [19] and CPI [20]. It is expected that new hardware technology such as Intel's Control-flow Enforcement Technology [21] will make these mitigations more efficient and robust against attacks.

7 Conclusion

In section 5.1 we observed that an attacker, despite lacking code injection capabilities, may still be able to execute arbitrary payloads in the context of a compromised process. Existing arbitrary code mitigations, therefore, do not prevent exploitation, but they attempt to make exploits more expensive to write by forcing the attacker to use code reuse techniques. In sections 5.2 and 5.3, however, we studied mechanisms that an attacker may use to keep exploit development costs low even when lacking code injection capabilities. We observed that, in the contexts in which RPCRAs are possible, attackers may build complex payloads from simple, reusable ROP sequences. In section 6, we studied exploitation flows that an attacker could still follow to compromise a target system. We observed that, if least privilege is not properly enforced, an attacker may still be able to propagate, achieving full system compromise or being able to utilize preexisting injectable payloads that should have been neutralized by code injection mitigations. We also observed that RPCRAs could be used by attackers to make border processes act as windows into the system; activity and network monitoring may become more relevant to identify such attacks. While the proof of concept only deals with ROP in a stack overflow scenario, it is still left to see whether the techniques may be applied in different exploitation contexts, with different code reuse techniques. However, given that an attacker may still have as much power as before, and that performing RPCRAs may not require primitives more complex that those typically acquired by successful exploits today, we may conclude that mitigations that attempt to prevent exploitation by preventing execution of injected code should not be a significant obstacle, and they should not increase exploit development costs significantly in the long run. More effort should be made to improve mitigations that attempt to prevent memory corruption and control flow hijacking in the first place, and effort should be made to enforce least privilege where possible.

8 Further Research

While the proof of concept only addresses the feasibility of the proposed techniques in a simple, laboratory scenario, it must be noticed that the threat model does not grant the attacker any capability that is not routinely abused by typical exploits today. It is left to see, however, an injector being developed for a large, productive piece of software. More interestingly, it is left to see if injectors can be developed in alternative exploitation scenarios that involve different exploitation techniques such as JOP [11] or COOP [12]. Since payloads based on these techniques tend to be more difficult to write than regular ROP payloads, they may also benefit from the simplicity and modularity introduced by the adoption of an RPCRA based scheme.

9 Acknowledgements

This research is supported by Fundación Sadosky, Argentina. Special thanks to Juan Martín Heguiabehere, head of STIC, for his review and support, and for providing constant insight and expertise to guide the team. Many thanks also to Herman Schinca of STIC for proofreading and providing detailed advice to improve the writing. Any error or misinterpretation, however, is only my own, and not of the aforementioned people. I would also like to show my appreciation to everyone in STIC and Fundación Sadosky.

10 References

[1] CVE-2018-8373, Scripting Engine Memory Corruption Vulnerability. Aug. 2018. `https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-8373`.

[2] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. *In IEEE Symposium on Security and Privacy*, 2013.

[3] David Weston & Matt Miller. Windows 10 Mitigation Improvements. Presentation, BlackHat USA 2016, Las Vegas, US-NV, Aug. 4, 2016.

[4] Matt Miller. Mitigating arbitrary native code execution in Microsoft Edge. Windows Blogs (blog). Feb. 23, 2017. Accessed May 23, 2019. `https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution`.

[5] David Weston & Matt Miller. Microsoft's strategy and technology improvements toward mitigating arbitrary native code execution. Presentation, CanSecWest 2017, Vancouver, Canada, March, 2017.

[6] Ivan Fratric. Bypassing Mitigations by Attacking JIT Server in Microsoft Edge. Google Project Zero, 2018.

[7] MSDN. SetProcessMitigationPolicy function, ProcessDynamicCodePolicy. Accessed May 30, 2019. `https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setprocessmitigationpolicy`.

[8] Bruce Jay Neslon. Remote Procedure Call. PARC CSL-81-9 (Also CMU-CS-81-119). Xerox Palo Alto Research Center. PhD thesis. May, 1981.

[9] Maximiliano Caceres. Syscall Proxying - Simulating remote execution. Core Security. 2002.

[10] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007, pages 552–61. ACM Press*, Oct. 2007.

[11] Tyler Bletsch, Xuxian Jiang & Vince W. Freeh and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symp. on Info., Computer and Communications Security*, 2011.

[12] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, & Thorsten Holz. Counterfeitobject-oriented programming. In *Proceedings of IEEE S&P*, 2015.

[13] PaX Team, Address space layout randomization (ASLR), 2003, pax.grsecurity.net/docs/aslr.txt.

[14] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lach-mund, & T. Walter. Breaking the memory secrecy assumption. In *EUROSEC'09*.

[15] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, & Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[16] Nicholas Carlini & David Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[17] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine, 49(14)*, Nov. 1996. `http://www.phrack.org/archives/49/P49-14`.

[18] N. Hardy. The confused deputy. *ACM Operating Systems Review*, 22(4):36–38, 1988.

[19] Martín Abadi, Mihai Budiu, Úlfar Erlingsson & Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computerand Communications Security*, November 2005.

[20] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[21] Intel. Control-flow Enforcement Technology Specification, Revision 3.0. May 2019.