COMPSCI 690G: Security for Large-Scale Systems
ROP-ing in different dimensions

Dung Nguyen
dung@umass.edu

May 15, 2024

1 Introduction

Return-oriented programming (ROP) has been long introduced in the security community, among both blue team and red team. In most use cases, it is a critical part of an exploit chain that leads to opening a shell or create an executable memory region in the targeted system. We aim to research improvements and develop techniques to further extent the possibility of ROP on x86 Instruction Set Architecture. In the scope of this paper, we explore ROP based on two possible ways of code execution that have rarely been studied in the past: redundant instruction prefix and Windows WoW64 subsystem.

2 Background

x86 Instruction Set Architecture (ISA) was designed by Intel based on its Intel 8086 16-bit microprocessor. Over more than 40 years, the ISA has evolved to support 32-bit (x86-32) and 64-bit (x86-64). Several interesting characteristics still remain today are variable encoding instructions, instruction prefix, and memory segmentation.

Since the introduction of W^X (Write XOR Execute), ROP embarked as a technique that writes no code but instead jump to pieces of code in the executable region of memory to execute. The idea is based on the fact that when we call a function, the address right after the instruction will be saved so that we can deterministically return after we finish executing the function, which is why it is also called *return address*. If we can override the return address, we can virtually control the instruction pointer. For ROP, we replace the return address with *ROP gadget*, a small piece of code that minimally changes the machine state and ends with a branching instruction. Using a series of gadgets - *ROP chain*, we can fake a history of function calls and execute a sequence of instructions that we want. This method is successful in x86 particularly because the ISA uses variable encoding for instructions, meaning a simple offset in decoding can create a new interpretation of instructions. There are two types of branch in x86: forward-edge, which directly modify program counter through CALL, JMP instruction; and backward-edge, which pop a value from the stack and change return address to it. For simplicity, this paper will focus on backward-edge type.

In the process of upgrading the ISA, prefixes were used to support new memory size and features while keeping backward-compatibility. Through observation, the frontend of Intel x86 CPU responsible for decoding instruction is relaxed, which means it simply ignores prefix that does not make sense or conflicts with other prefixes[1]. In other words, the CPU would execute as if without the prefix. For such cases, we call it *redundant instruction prefix*.

Windows on Windows (WoW) is a Windows compatibilty layer technology that provides support for the OS to run software built for architectures different from the native machine's architecture, such as running x86-16 code on x86-32 machine, x86-32 code on x86-64 machine, or more recently, x86-64 on 64-bit ARM machine. *Windows WoW64* specifically refers to the x86-32 subsystem on 64-bit version of Windows. Per documentation, this is based on the Long Mode feature of x86-64 CPU, where it is possible to switch between compatibility mode (x86-32) and 64-bit mode (x86-64) by flipping CS.L bit. Windows repurposes the memory segmentation feature to implement transitioning between x86-32 and x86-64 in WoW64. This environment is only initialized when user executes a 32-bit binary on Windows 64-bit version, where the operating system will additionally load three Dynamic Link Libraries (DLL) to assist the transition: `wow64.dll`, `wow64cpu.dll`, `wow64win.dll`.

As a convention, from here on out, this paper will use 'x86' to refer to x86-32 ISA and 'x64' to refer to x86-64 ISA.

3 Problem

Many solutions preventing ROP and guaranteeing integrity of control-flow are being widely deployed. Among them are software-based solutions like Control-Flow Integrity in Clang compiler and Control-Flow Guard on Windows, which insert instrumentations verifying the validity of call target and inhibit forward-edge ROP gadgets. At the same time, modern malware scanner and anti-virus hook on critical functions and raises alert when they are suspiciously called. Having better chance here is the use of backward-edge ROP gadgets, which utilizes RET instruction in x86 instruction set architecture. We devise two problems to analyze in this paper:

1. Using redundant instruction prefix, can we enrich our gadget database in x86/x64?

2. Through Windows WoW64 subsystem, can we increase the number and reliability of ROP gadgets to arbitrarily execute code on the target Windows machine?

4 Prior works

Within the cybersecurity community, searching for gadgets is one of the most fundamental needs to build a ROP chain. There are at least three popular ROP gadget finders: ROPgadget [2], Ropper [3], and onegadget [4]. ROPgadget supports a wide range of architecture with numerous filter options, along with generating syscall ROP chain based on Write-What-Where primitive and mostly POP and INC instructions. Ropper additionally allows generating execve, mprotect, and virtualprotect ROP chain and uses Z3 to filter gadget by constraints. Meanwhile, onegadget is a special tool that finds single gadgets in GNU libc to open shell using symbolic execution. One common thing between these tools is the detection and assumption of the architecture based on the binary metadata.

Nicholas Carlini and David Wagner in 2014 has researched on several advanced techniques for ROP attack that break CFI-based defenses [5]. They identify possible sequence of gadgets that hide history, perform evasion, etc. and defeats kBouncer, ROPecker. Q system by Schwartz et al. is a ROP compiler that use SMP solver to produce more concrete, diverse exploit payload as well as hardening payload that crashes with defense on [6].

In the past, for Windows shellcoding, there exists tools that generate Windows syscall such as SysWhispers, and building the shellcode targeting specific version of Windows such as Dumperts [7]. These tools only targets native environment, such as it will only generate x64 shellcode for Windows x64. But more recently, there is a development of a tool to automatically build shellcode that hooks 64-bit code in WoW64 32-bit mode in the payload called ShellWasp[8]. In ShellWasp version 2.0, it includes several well-developed techniques that aid with calling Windows API, such as Hell's Gate, Halo's Gate, Heaven's Gate. Interestingly, the author of ShellWasp is currently working on ROP_ROCKET, an experimental ROP chain generator that can construct calls to LoadLibrary and GetProcAddress on native system, or exclusively apply Heaven's Gate. Our research focuses instead on expanding capabilities after going through Heaven's Gate.

Heaven's Gate is a chain of techniques that executes x64 code within Windows WoW64 subsystem. It was discovered as early as 2009 by rgb/defjam, less than a year after the release of Windows 7 that first introduces Windows WoW64. This feature is designed with backward compatibilty in mind, but it also provides us a possibility to decode bytes and execute instructions differently since the executable region is shared between the native and WoW64 subsystem. The followings are main stages of Heaven's Gate [9]:

1. From 32-bit executable, perform a far jump similar to the gate found in `wow64cpu.dll!X86SwitchTo64BitMode` that leads us to 64-bit code, 64-bit address space, 64-bit registers, and 64-bit calls. The segment selector for the native 64-bit mode is 0x33, and 32-bit compatibility mode is 0x23 (which we are already in).

2. Enumerate imported modules for `ntdll.dll`. This can be achieved by accessing PEB64 (Process Environment Block). By the design, there are seperate PEB and PEB64 when running process in WoW64.

3. Knowing where `ntdll.dll` in memory, we can infer the Windows API functions and proceed to call any of them.

We expect to perform similar execution with ROP.

5 Methodology

For redundant instruction prefix, we will first probe the possibility of applying the technique in x64 binary. If the chance is likely, this work will either reuse silifuzz that was used in the work of reptar[1], or develop a new fuzzing framework that can target our special environment.

For WoW64, to create ROP with higher chance of success, we examine and apply methods found in Nicholas Carlini and David Wagner's ROP gadget and Q system by Schwartz et al. To materialize WoW64 access gadget, this work will cross-refer WoW64 techniques implemented in SysWasp 2.0 [8] and ROP_ROCKET [10] by Bramwell Brizendine.

By experimenting with Windows WoW64 with ROP, we aim to answer the following questions:

- What are the hard requirements of WoW64 with ROP?
- How flexibility the method of going to and returning from WoW64 is?
- Can we combine Heaven's Gate with ROP chain?

To verify the ROP chain, we use pwntools commit `8ba1bdf0b6fe` which includes support for executing Windows binary and debugging using WinDbg. We use classic version of WinDbg that comes with Windows Software Development Kit (SDK) to test and speculate states during ROP chain execution. The WinDbg Preview available in Microsoft Store (also known as WinDbgX) does

not support the transition of Windows WoW64, so using it is not recommended.

As part of the research, a tool named *sirop* would be created to search for necessary gadgets and generate a complete ROP chain. Eventually, the result paper will document the formalized methodology exercised throughout the paper. All versions of software are detailed in the **Appendix**.

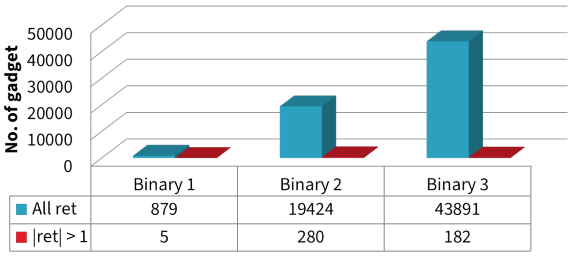
6 Result

6.1 Detecting ROP

In this section, we provide formal explanation for searching a complete ROP gadget. It is advised to also follow C++ snippets in the Appendix section and source code. We first search for branching instructions in executable memory region. For example, `C3` for RET and `CB` for RET FAR. `try_decode` takes in branching instruction location and the length of extra bytes if the instruction is longer than 1. It first guess possible size of gadget by subtracting from 0 to `max_size` set by user. Then for every pair of gadget size and architecture, using `decode_offset`, we attempt to decode at the subtracted offset. `complete_ret` will filter complete and valid gadget, where it must have only one branching instruction and left out no extra bytes at the end of its disassembly.

6.2 Redundant instruction prefix

As described in the background, x64 CPU is not strict in decoding instruction prefixes. While this idea initially sounds particularly fascinating, our result shows that the redundant prefix does not appear asin real binary. We experiment this by searching for RET instructions including ones with any additional prefix(es). We modified the algorithm described in previous section to seperate RET with and without redundant prefix. The figure below shows the number of such gadgets based on our run of the tool on three selected binary files. As such, it is shown to be insignificant.



	Binary 1	Binary 2	Binary 3
All ret	879	19424	43891
|ret| > 1	5	280	182

We realized that redundant instruction prefix has already been playing a part in creating ROP gadgets. For ROP gadget search tools listed in **Prior works**, most of them capture these kind of gadgets without implicitly disclose this fact partly because most disassemblers in them would also ignore the redundant prefix when decoding. However, we do not know if the disassemblers are consistent with CPU front-end or not. Since this is an inefficient approach, we did not pursue the search further.

6.3 Windows WoW64

Contrary to the previous approach, we have a lot more opportunities for this. To execute and verify the capability of performing ROP in this specific kind of environment, we choose to reimplement Heaven's Gate.

6.3.1 Heaven's Gate

To switch from 32-bit WoW64 to 64-bit native is easy. We find gadget `RET FAR` (1-byte instruction `CB`) is the easiest to search for and execute. Per both Intel documentation [11], both x86 and x64 versions of `CB` function similarly: popping 2 32-bit values from the stack for parameters. In **Appendix**, we provide recipe for this successful transition. It is also observed that stacking multiple payload is possible, meaning there are no visible hinders that impact switching freely between the two environments.

To acquire PEB64, we tested and validated two possible methods that is still working up until Windows 11:

- Read from TIB: Windows 64-bit stores TIB in GS segments, so we need an instruction that read 32-bit values from GS:[0x60]. With shellcode, we can do `mov rax, GS:[0x60]`. This is hard to achieve particularly because of the size itself (8 bytes, `65 67 48 A1 60 00 00 00`) and the unavoidable null bytes. We try to scan the 32-bit ntdll.dll and found some prominent alternatives:
  - sbb edi, gs:[rax+0x6E]; ret
  - fild gs:[rcx]; ret
  - fidiv gs:[rdi+0x8E]; add ch, al; mov rax, rdx; add [rax], al; add [rax], al; push rsp; ret
- r12 register: As observed by several informal documentations, for unclear reasons, R12 register also contain the address of PEB64 [12]. This way is easier to achieve, but the appearance of gadget to read r12 is incredibly rare.

After this point, we attempted to locate ntdll.dll for calling Windows API, but we failed to gather enough gadgets to traverse the loaded module list. From Windows 8, ntdll.dll is loaded at address higher than 2^{32}, meaning we cannot simply store it in a 32-bit register [13, 12]. Such work is possible given enough gadgets and incredible amount time. During our gathering of ROP gadgets,

among 'extra' gadgets produced due to different interpretation of x64, many of them are found to be duplication of x86 gadgets. As this also depends on the way compilers generate code on different optimizations and switches, we leave this point of unknown for future research.

Note that this is not the first time the technique is executed, so the above experiment was done with several assumption on thread model. First, the attacker must know the Windows version as system call regularly changes between versions. To access the address of ROP gadgets, base memory address must be known when ASLR is turned on, or ASLR must be forcefully disabled. If the executable also employs stack canary, a bypass is assumed to exist.

6.4 Tools

Most of the current tools assume a single architecture based on executable metadata, which is unfitting in our case. Therefore, we have developed a scanning tool to assist in searching possible gadgets called **sirop**. Written in C++23, it uses Zydis to disassemble both x86-32 and x86-64 architecture. For access to full source code, please see **Appendix** section.

7 Conclusion

Among the two approaches we research, the more successful one is Windows WoW64, although a careful selection of target binary and dedicated amount of time are still needed to foster a complete ROP chain. While it was mentioned that the technique might introduce more gadgets [12], through speculation using our tool, most of newer gadget are duplications. We believe this is inherently due to the design of x64 as a rough superset of x86. In any case, the Windows WoW64 can be used to hide processing from security tools, as many of these cannot hook into this environment or properly disassemble the code after we go through the Heaven's Gate.

While the capability extension is up to expectation, along with the fact that Microsoft has patched many holes in WoW64 [13], we hope that this research still holds value as a point of reference for evasion technique for ROP. Much more recently, this mechanism is reworked to additionally support ARM and ARM64 on Windows 10 on ARM and Windows 11 on ARM, respectively. A new kind of binary Arm64X containing both x64 and ARM64 code is introduced, which may or may not hinder the procress of exploitation.

References

[1] Tavis Ormandy. *reptar*. URL: https://lock.cmpxchg8b.com/reptar.html.

[2] Jonathan Salwan et al. *JonathanSalwan/ROPgadget*. Sept. 1, 2023. URL: https://github.com/JonathanSalwan/ROPgadget.

[3] Sascha Schirra et al. *sashs/Ropper*. Jan. 11, 2024. URL: https://github.com/sashs/Ropper.

[4] david942j et al. *david942j/one_gadget*. Mar. 2, 2024. URL: https://github.com/david942j/one_gadget.

[5] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[6] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "Q: Exploit Hardening Made Easy". In: *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011. URL: https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy.

[7] Bramwell Brizendine and Tarek Abdelmotaleb. "Syscall Shellcode in WoW64 Windows". In: 2022. URL: https://api.semanticscholar.org/CorpusID:252276833.

[8] Bramwell Brizendine. *Windows Syscalls in Shellcode: Advanced Techniques for Malicious Functionality*. Hack In The Box. Apr. 2023. URL: https://conference.hitb.org/hitbsecconf2023ams/session/windows-syscalls-in-shellcode-advanced-techniques-for-malicious-functionality/.

[9] roy g biv / defjam. *Heaven's Gate*. Apr. 1, 2011. URL: https://archive.is/5Rkr2.

[10] Bramwell Brizendine and Shiva Shashank Kusuma. *Bw3ll/ROP_ROCKET*. Apr. 7, 2024. URL: https://github.com/Bw3ll/ROP_ROCKET.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Apr. 24, 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[12] Darren Kemp and Mikhail Davidov. *WoW64 and So Can You*. Nov. 2, 2015. URL: https://duo.com/assets/pdf/wow-64-and-so-can-you.pdf.

[13] Alex Ionescu. *Closing "Heaven's Gate"*. Dec. 30, 2015. URL: https://www.alex-ionescu.com/closing-heavens-gate/.

A Appendix

A.1 Source code

For access to full released source code, please visit the following link (require UMass account).
`https://drive.google.com/file/d/13LXqnwwKX6WjR3bgij9TJZ1cqy8P2G1m/view?usp=sharing`

A.2 Version table

Software	Version
Windows OS	11 23H2
Windows SDK	10.0.22621.2428
MSVC	19.39.33523
Clang	18.1.5
Python	3.12.1
CMake	3.29.0-rc2
Pwntools	4.14.0-dev (8ba1bdf0b6fe)
Zydis	4.1.0
LIEF	0.14.1

A.3 ROP searching algorithm in C++23

This is an excerpt from actual source code mentioned above.

```
auto try_decode = [&](const std::pair<uint64_t, uint8_t> p) -> auto {
    const auto [idx, op] = p;
    const auto right_off = rop_seed.at(op);
    auto decode_offset
        = [&exec_content, va, right_off,
           idx](std::tuple<uint64_t, ZydisMachineMode> args) -> auto {
        auto [x, mode] = args;
        return std::make_tuple(
            decode_x86_zydis(std::ranges::ref_view{ exec_content }
                             | views::drop(idx - x)
                             | views::take(x + 1 + right_off),
                         va + idx - x, mode),
            mode);
    };
    auto complete_ret = [](auto x) {
        auto &[gadget, _] = x;
        auto &[v, offset, rem] = gadget;
        auto count = std::ranges::filter_view(
            v, [](const ZydisDisassembledInstruction i) -> bool {
                return is_branch(i);
            });
        // The last RET should be the only one
        return rem == 0 && !count.empty()
            && (std::addressof(count.front()) == std::addressof(v.back()));
    };

    auto range
        = views::iota(0ULL, std::min(static_cast<uint64_t>(max_size), idx));
    return views::cartesian_product(range, machine_mode)
         | views::transform(decode_offset) | views::filter(complete_ret);
};
```

A.4 Windows WoW64 basic recipe

A.4.1 Target binary: bof.c

```c
#include <stdio.h>
#include <string.h>
// clang -m32 bof.c -o bof.exe

void win() { puts("You Win!"); }

void feedback() {
    char s[0x10] = { 0 };
    printf("%s ", "Please enter your feedback:");
    fgets(s, 0x1000, stdin);
    puts("Thank you! Your feedback will be processed");
    printf("Here is the feedback hash: %0lx\n", *(long*)s);
}

int main() {
    printf("%p\n", &main);
    feedback();
    return 0;
}

__attribute__((constructor)) void fixbuf(void) {
    setvbuf(stdin, (char *)NULL, _IONBF, BUFSIZ);
    setvbuf(stdout, (char *)NULL, _IONBF, BUFSIZ);
    setvbuf(stderr, (char *)NULL, _IONBF, BUFSIZ);
}
```

A.4.2 pwntool script

```python
from pwn import *

target = "test\\bof.v2.exe"

context.os = "windows"
context.arch = "i386"
if args.WINDBG:
    r = windbg.debug(target, windbgscript="bp $exentry", exe=target)
else:
    r = process(target)

main2base = -0xB0
main_addr = int(r.recvline(), 16)
base = main_addr + main2base
win = base
r.recvuntil(b"Please enter your feedback:")
print("main address: 0x{:x}".format(main_addr))
original = base + 0xDA
payload_m1 = (
    b"A" * 20
    + p32(base + 0xFEE)      # [x86] ret far
    + p32(base + 0x148a)     # [x64] 0x148a: xor eax, eax; ret
    + p32(0x33)
```

```
    + p64(base + 0x12B5)    # [x64] 0x12b5: add eax, 0x14; ret
    + p64(base + 0x12AC)    # [x64] 0x12ac: add eax, 0x10; ret
    + p64(base + 0xFEE)     # [x64] ret far
    + p32(base + 0x91)      # printf
    + p32(0x23)
    + b'Z' * 0x20
)

r.sendline(payload_m1)

print("Starting interactive mode ...")
r.interactive()
```