ROP Compiler

Jeff Stewart, Veer Dedhia

I. Introduction

When developing exploits for modern x86 64-bit systems, attackers must handcraft exploits for each binary. This involves finding a vulnerability (such as a stack-based buffer overflow) and diverting control flow (overwrite return address). Modern exploits employ Return-Oriented Programming (ROP) to bypass widely deployed defenses such as W^X.

Building a ROP chain requires manual effort to find suitable gadgets out of the multitude of existing code snippets, and then chain those gadgets together in the correct order to call functions or execute injected code. x86 64-bit systems present some challenges that do not exist on other platforms. For example, the 64-bit calling convention primarily uses register arguments, as opposed to stack-pushed arguments on many 32-bit systems. This requires finding gadgets to set values in registers, instead of using an overflow to write to the stack. While many tools exist to help the various stages of exploit building, no public compiler is available to fully create these ROP chains.

We present a simple ROP Compiler, developed to more easily generate ROP chains, given a binary and goal. We demonstrate our compiler on both a proof-of-concept simple binary, as well as a well-known utility, rsync. Our tool generates working ROP chains to inject and execute shellcode or call other functions.

A. Background

Control flow hijacking is an extremely dangerous attack vector. They allow an attacker to divert the intended program flow, and instead execute instructions that further the attacker's goals. These goals may include spawning a shell or installing malware. This gives the attacker a foothold on the machine towards a larger goals, such as financial fraud or exfiltration of sensitive data.

Modern computer systems employ various defenses, such as W^X and ASLR, that protect against different types of control flow hijacking. W^X works by disabling execution of writable memory, such as the stack. This prevents an attacker from injecting malicious code as user-supplied input and directly diverting control flow to that address.

ASLR (address space layout randomization) works by loading a given binary at a random offset in memory for each execution. Libraries (libc, etc.) use position-independent code (PIC), so that they may be shared amongst processes or loaded separately. Binaries that support ALSR are known as position-independent executables (PIE), which means that the code does not use hardcoded addresses can be loaded at any address. Non-PIE binaries have hardcoded addresses, and thus they do not support ASLR.

With these defenses widely deployed, attackers have turned away from simpler techniques (code injection), and have instead developed control hijacking that reuses existing program code. This easily bypasses W^X, since the code is already allowed to execute. Libraries, such as libc, provide many functions that an attacker would like to call (execve, mprotect, etc.). As such, the first techniques focused on setting up stack arguments and then diverting control to a library function (return-to-libc).

However, this technique is limited to calling a single function, whose arguments come from the stack. An attack goal may require calls to functions whose arguments come from registers or setting up other prerequisites, however, so chaining becomes the necessary next step. Return-Oriented Programming (ROP) [6] is a technique developed to generalize the return-to-libc attacks by chaining snippets of code that end in 'ret' (called gadgets). Most modern exploits leverage ROP to bypass existing defenses.

There do exist some tools that find gadgets and suggest ROP chains, such as Q [5], ROPC [2], mona.py [1], and ROPgadget [4]. However, these tools are either not available (Q), do not work on realistic binaries (ROPC), x86 64-bit binaries (mona.py), or are not fully featured (ROPgadget).

II. Implementation Details

Our ROP Compiler is separated into four main components: the goal interpreter, the gadget finder, the gadget classifier, and the gadget scheduler.

A. Goal Interpreter

The goal interpreter provides the necessary configuration data for the ROP compiler. This configuration data is supplied by the exploit author and is comprised of three separate components:

1) A list of files to extract gadgets from. For PIE executables and libraries, each file should include the address that it was loaded to.
2) A list of libraries to extract metadata from. This list of libraries will be used to obtain function metadata for use later, as described in Section II-D.
3) A list of the author's desired goals. Each goal item can be one of three types: *Function*, *ShellcodeAddress*, *Shellcode*. *Function* goals indicate the author wishes to call a given function with a set of specified arguments. *ShellcodeAddress* goals indicate the author wishes to run shellcode that already exists in the target program's address space. The *Shellcode* goal indicates the author wishes to run a set of shellcode that does not already exist in the target program's address space. Thus, the

Name	Input	Parameters	Semantic Definition
JumpG	AddrReg		RIP $\leftarrow$ AddrReg
MoveRegG	InReg, OutReg		OutReg $\leftarrow$ InReg
LoadConstG	OutReg	Value	OutReg $\leftarrow$ Value
ArithmeticG	InReg1, InReg2, OutReg	$\diamond_b$	OutReg $\leftarrow$ InReg1 $\diamond_b$ InReg2
LoadMemG	AddrReg, OutReg	Offset	OutReg $\leftarrow$ M[AddrReg + Offset]
StoreMemG	AddrReg, InReg	Offset	M[AddrReg + Offset] $\leftarrow$ InReg
ArithmeticLoadG	AddrReg, OutReg	Offset, $\diamond_b$	OutReg $\diamond_b\leftarrow$ M[AddrReg + Offset]
ArithmeticStoreG	AddrReg, InReg	Offset, $\diamond_b$	M[AddrReg + Offset] $\diamond_b\leftarrow$ InReg

TABLE I: Gadget Types that our Gadget Classifier can find. M[addr] means accessing memory at the address addr and $\diamond_b$ means an arbitrary binary operation.

Shellcode goal must first load the shellcode into memory before it may continue.

These 3 goals were chosen to allow the exploit author to obtain arbitrary code execution, while also specifying the minimum amount of work that is necessary. For instance, there is no need to use a *Shellcode* goal if the exploit author already knows the address of their shellcode in memory.

B. Gadget Finder

The gadget finder iterates over all of the available binaries and searches for gadgets. This is accomplished by stepping through each byte (not instruction) of the executable sections of each binary, and then trying to disassemble the last **N** number of bytes. If a series of bytes ends in a RET or JMP instruction, then that series of bytes is saved as a potential gadget. For disassembly, our ROP compiler makes use of the Capstone framework [3].

The choice of **N** is a compromise between the speed of the ROP compiling process, and the number of gadgets found. As larger gadgets tend to be more complex (and thus less useful during ROP compilation), increasing **N** is not necessarily the best option. Our implementation makes the compromise of setting **N** to 10. A future version may iteratively increase **N** upon failing to compile an adequate ROP chain.

C. Gadget Classifier

The gadget classifier analyzes each of potential gadget to determine its type(s). Our classifier's implementation is based on the design of Q's classifier as presented by Schwartz et al. [5]. Schwartz proposed classifying gadget into 9 types useful for generating ROP chains. Our implementation classifies our potential gadgets into the 8 gadget types shown in Table I (excluding NoOpG gadgets from [5]).

Similar to the classifier proposed by Schwartz, our implementation uses the weakest precondition technique. For each gadget, we generate a symbolic formula for the instructions. This formula is then used to concretely emulate the gadget by substituting random values for any initial registers used and memory reads. The output registers and memory writes are then used to evaluate which, if any, of the gadget type preconditions hold true. This process is then done a number of times. Any preconditions left unviolated throughout all of the concrete executions reveal the gadget type.

While Q [5] and ROPC [2] take this process one step further, by utilizing a SMT solver to verify the gadgets; our implementation does not do so. In our brief analysis and examples (described in Section III), we have not seen a gadget that has been misclassified using only the concrete emulation. Thus, to improve the ROP compilation speed, our implementation does not verify gadgets. Any of the necessary metadata (such as which registers are clobbered by a gadget) is extracted from the symbolic formula.

D. Gadget Scheduler

The gadget scheduler combines classified gadgets into the individual goals. These goals are accomplished by iterating over the classified gadgets to find a sequence that match the required goal. For example, to write a value to memory, the scheduler looks for a LoadMemG gadget to set the address, another LoadMemG gadget to set the value, and a StoreMemG to perform the write to memory. The scheduler keeps track of the registers to ensure that one used in the beginning of a chain does not clobber one needed later in the chain.

While the *Function* and *ShellcodeAddress* goals are relatively straight forward to implement, the *Shellcode* goal requires more work. In order to execute the exploit author's shellcode, our scheduler first must write the shellcode to memory. This is accomplished by querying the goal interpreter for a writable section of memory, and then chaining together a series of write memory chains as previously described. Once the shellcode has been stored to memory, the *Shellcode* goal can be accomplished through the same means as a *ShellcodeAddress* goal.

A final situation that requires additional work is calling external functions in libraries. While calling functions within the main binary's Procedure Linkage Table (PLT) is relatively simple, calling a function not imported into the PLT requires substantially more work. To accomplish this task, the scheduler must first read from the target process's GOT for a function which the target process does use. Next, the scheduler adds the offset from this base function to the target function in the library. As offsets within a library are constant under traditional ASLR, the computed value will be the address of the target function. Our scheduler implements this approach by finding a chain of the following gadgets:

1) LoadMemG from the stack to set the address of the base entry to read in the GOT
2) LoadMemG to read the base entry in the GOT
3) LoadMemG from the stack to set the offset from the base entry to the target entry in the library
4) ArithmeticG to add the offset from the base entry to the target entry in the GOT

5) One LoadMemG per argument for the target function to set the argument's value
6) JumpG to jump to the function in the library

The scheduler scans for each gadget while making sure any values used in the later gadgets (such as the computed function's address) aren't clobbered by the earlier gadgets.

III. Example Usage

A. Buffer Overflow Proof of Concepts

In order to provide a test suite for our ROP compiler, we've implemented a series of proof of concept stack buffer overflow exploits. These example proof of concept programs exercise the different components of the compiler. This test suite includes:

1) A buffer overflow which calls mprotect from the PLT to change memory permissions and execute shellcode already existing in memory
2) A buffer overflow which calls the syscall function from the PLT to change memory permissions and execute shellcode already existing in memory
3) A buffer overflow which calls mprotect from the PLT to change memory permissions and execute shellcode not already existing in memory
4) A buffer overflow which calls system from the PLT to run a command
5) A buffer overflow which reads the address of printf in the GOT, adds the offset from printf to mprotect in libc, calls mprotect to change memory permissions, and executes shellcode not already existing in memory

B. rsync

In order to illustrate a real world use of our ROP compiler, we implemented an exploit for rsync. As the current rsync[1] does not have any publicly known vulnerabilities, we introduced a synthetic stack buffer overflow into rsync. This stack buffer overflow vulnerability was introduced into rsync's filter file argument processing code. rsync provides the ability to filter the files that are copied through the use of a filter file that specifies which files to exclude. As this vulnerability is a file format bug, there is little chance for the attacker to leak information about the memory layout of rsync (as compared to a network reachable vulnerability). Thus, a working exploit must be able to load shellcode at a known address. The exploit is further complicated by the need to call mprotect without it being in rsync's PLT.

Our exploit, provided in *rsync.py*, utilizes an automatically generated ROP chain that handles these issues. The resulting ROP chain is 464 bytes that loads and executes a 33 byte execve-based shellcode. This ROP chain was generated in 71 seconds on the author's laptop with an Intel i7-4700HG CPU and 8 GB of RAM.

IV. Future Work

Our ROP compiler, while functional, could be improved to provide additional advanced features. For example, we would like to explore synthesizing new gadgets by combining existing smaller gadgets. This would enable our compiler to produce chains on smaller binaries. We would also like to expand our attack goal language to allow for more expressive goals. While Turing completeness was abandoned early on, in terms of practicality, a more expressive language would allow more complex exploits, such as those needed in environments that do not allow remapping of memory permissions [7].

References

[1] Corelan. Mona. https://github.com/corelan/mona, 2015.

[2] pakt. Ropc - a turing complete rop compiler. https://github.com/pakt/ropc, 2013.

[3] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. http://www.capstone-engine.org/BHUSA2014-capstone.pdf, 2014.

[4] Jonathan Salwan. Ropgadget. https://github.com/JonathanSalwan/ROPgadget, 2015.

[5] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[6] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[7] Brad Spengler. grsecurity: Features. https://grsecurity.net/features.php#mprotect, 2015.

[1] rsync 3.1.1, current as of 12/6/15