The Rop Mate

Visually Assisting the Creation of ROPChain based Exploits

Visual Analytics - Sapienza

Pietro Borrello Serena Ferracci

Abstract

ROPChain based exploits are increasingly present in the scenario of advanced attack techniques. Being able to test software against such types of attacks is fundamental to any RedTeam hacker. Automatic tool that try to create ROPChains exists and are well known, but they often fail with non usual binaries. In this paper we propose `RopMate`, a visually aiding tool to help the human ROPChain builder to create its ROPChain. We strongly believe that there is the need of such a tool, since the existing state of the art, only propose text based approach with, less or more, informative lists of the gadgets founded. RopMate presents the builder with a clear interface of available gadgets, in which he is able to choose which of them to add to the building ROPChain, and, once selected, to filter the list according to wanted guarantees on preserved registers and accessed memory. The user is also able to find similar gadget in respect with the one chosen.

1 Introduction

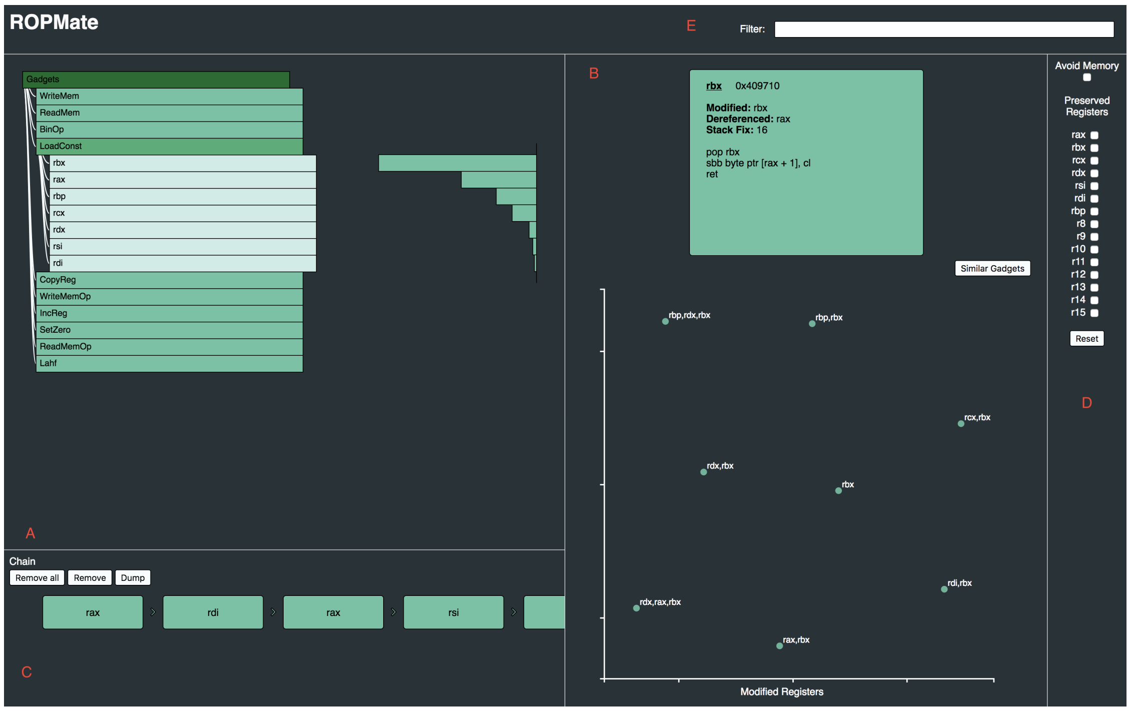
Return-oriented programming (ROP) is an exploiting technique that allows an attacker to induce arbitrary behavior in a vulnerable program without actually injecting any code, but through a chain of redirections in the program memory itself. [1]

The attack scenario is based on a controlled stack frame, where the return address can be overwritten. This is an advanced version of the "return to libc attack" [2] where multiple pieces of code are called in sequence to provide the needed code semantics execution: the attack combines a large number of short instruction sequences (called *gadgets* from now on) that allow arbitrary computation. This is resilient to mitigation as non executable memory areas.

Each gadget is in the form of a couple of instruction followed by a return. This allows the attacker to place a sequence of gadget addresses on the stack (called *ROPChain*) from the return address on, that will be executed thanks to the semantics of the ret instruction.

In this paper we aim to present a first step toward the acceptance that automatic tools will never be perfect, therefore human intervention will always be needed. Our tool is meant to give an help to any human ROPChain builder, in order to ease the creation of part or the totality of the chain.

While previous tools like, ropper [3] or ROPGadget [4], do an extraordinary job in finding useful gadgets, they lack totally of user friendliness. They present the chain builder with a plain text dump of gadget found, that is meant



```
rbx    0x409710

Modified: rbx
Dereferenced: rax
Stack Fix: 16

pop rbx
sbb byte ptr [rax + 1], cl
ret
```

Figure 1: ROPMate

to be parsed by and or scripting. While this is useful for other automatic tools, presenting the users with thousands of lines of gadgets doesn't help him in any way on the choice of the right gadget. Our work focuses on making the life of the exploiter easier.

2 The Problem

There exists automatic tools to find the suitable gadgets to create the final chain, but all of these tools present the attacker with a textual list of all the found gadgets. Then the attacker has to filter that list on the terminal with `grep` or similar commands. This becomes frustrating and infeasible with the growing of the complexity of the needed chain.

This is an example of the most simple ROPChain to spawn a shell in the libc standard library:

```
rop = ''
rop += p64(...) # pop rdi; ret;
rop += '/bin/sh\x00'
rop += p64(...) # pop rsi; ret;
rop += p64(WRITABLE_ADDRESS)
rop += p64(...) # mov [rsi], rdi;
    ret;
rop += p64(...) # pop rdi; ret;
rop += p64(WRITABLE_ADDRESS)
rop += p64(...) # pop rsi; ret;
rop += p64(0x0)
rop += p64(...) # pop rax; ret;
rop += p64(...)
rop += p64(...) # syscall; ret;
```

Where `p64()` transforms an address to the string representation of it in 64 bits, to be able to place it on the stack:

```
>>> p64(0x1122334455667788)
            '\x88\x77\x66\x55\x44\
               x33\x22\x11'
```

The comment near each instruction represents the instructions in that gadget to be executed.

The problem is that it never happens that the gadget are so simple and clear. Mixed with the instruction the attacker needs, there can be register modifications and memory accesses that can ruin the generated semantics, so choosing the next gadget to use is a delicate task. This becomes a pain, as the size of the chain grows and the number of constraints to maintain in mind increases. Moreover the number of gadget available in a binary is huge (from 1.500 for a middle sized binary to more than 15.000 for a standard library) so naively searching for useful and correct gadgets between them is impossible.

Automatic tools that generate the chain exist but usually they fail and leave the attacker alone. So there is a lack of an effective tool to help developing the exploit.

3 Related Works

Code Reuse Attacks. Memory corruption bugs for memory unsafe languages are one of the oldest problems in software security. Historically, attackers have exploited buffer overflows caused by coding errors to inject their own code into the application and have the instruction pointer jump to it. Data Execution Prevention techniques that are deployed on modern systems try to mitigate execution of malicious payloads that can be inserted. Therefore Attackers have switched their attention to create attacks reusing code already present in the binary, calling functions present in the linked standard library or chaining together short instruction sequences to carry out the desired computation. Return Oriented Programming is the most famous embodiment of the latter approach: by arranging the addresses of the code sequences to be executed on the stack along with their operands as part of a ROP chain, the `ret` instruction present at the end of each gadget instructs the CPU to follow the flow entailed by the chain.

ROP Gadget Finders. There exists a plethora of tools to find executable gadgets (sequence of executable and correct instructions that ends with a `ret` instruction): all of them provide the attacker with a list of gadgets among which to choose [3] [4].

Automatic ROPChain Builders. Most of the tools that provide some functionality to find gadgets, also try to automatically build a chain with some predefined desired effects. While they usually succeed with standard binaries (as `libc`) or standard chains, they often fail on unusual binaries, and don't provide any help to build custom chains.

Visual ROPChain Builder. At the time of the paper we weren't able to find any visual or textual tool, that would help an attacker to build his ROPChain. Current techniques just rely on python scripting.

4 Proposed Solution

We propose to apply Visual Analytics methods to the problem. The aim is to present an interface that will help the construction of the chain for the exploit. The binary that will be the source of the gadgets is analyzed by the backend server that produces a list of semantically meaningful gadgets. This means that only gadgets that have a clear effect are maintained.

The interface contains the list of all meaningful gadget, divided by class, and by effects on parameters, in which the user can make queries on the desired features and immediately visualize the gadgets that satisfy them. Queries may involve searching for gadgets that have a particular semantical meaning, but for example, don't modify some registers that have yet been set or access the memory only by some controlled registers, not to crash the program.

Clicking on a gadget will show its features (that will be encoded with additional visual hints). Each class of gadget has a set of attributes listed in the Table 1.

Each gadget has a specific type, between:

`LoadConst, SetZero, IncReg, CopyReg, BinOp, ReadMem, WriteMem, OpEsp, Lahf, ReadMemOp, WriteMemOp`

5 ROPMate

In this section we describe ROPMate, a prototype visual ROPChain builder, that assists the attacker in composing the chain with the desired semantics. ROPMate lets the user to filter and select gadgets to add to his chain, while presenting to him only the gadgets that fits the semantics and guarantees he wants. The user is also able to search for similar gadgets with respect to the selected one, needing the same semantics, but different guarantees.

5.1 Gadgets Dataset

The system takes ad input a dataset of semantically analyzed gadgets, provided by a backend server. The backend server essentially takes the original binary collects the gadget through some existing gadget finding tool, and performs a semantic analysis on each found gadget to set the type and the attributes described in Table 1.

5.2 User Interface

ROPMate is divided into five main views as shown in Figure 1: the *Tree View* (Figure 1A), the *Chain View*(Figure 1C), the *Analysis View*(Figure 1B), the *Control Panel* (Figure 1D) and the *Filter View* (Figure 1E).

The user will search for the wanted gadget in the *Tree View*, while inspecting the selected one and searching similar in the *Analysis View*. Once added, a gadget, will show up in the *Chain View*, and it can be further moved or removed from the chain. Filters in the *Control Panel* and *Filter View* can be user to restrict the search to useful gadgets.

5.2.1 Tree View

(Figure 1A)

The tree view can be considered the main view of the tool: it provides the whole list of analyzed gadgets. Filters and controls apply to such list to restrict the search space. The gadgets in the main view are hierarchically ordered by class, and then by parameters that represents their semantic meaning. The list of categories reported in the previous section. In the last level

Table 1: Gadget attributes

Gadget	
type	the class of the gadget
params	the parameters of the gadget
hex	the x86 bytes instructions in the gadgets
disasm	the x86 disassembled instructions
address	the address where the gadget starts in memory
address_end	the address where the gadget ends in memory
modified_regs	the registers modified by the execution of the gadget
mem	the registers that the gadget uses to access memory
stack_fix	the delta in the stack after the gadget execution
retn	how many words the ret instruction at the end pops out

of the tree the gadgets are presented, displaying the assembly code of each gadget. Since gadgets are aggregated by semantical meaning displaying the assembly code is the only way to distinguish them at first sight.

Near an opened category, an histogram is displayed, indicating to the user the number of gadgets of each class, to quickly give an hint on the overall view. Our first design choice of integrating the histogram in the tree, just lead to an ugly and unclear interface, therefore we chose to divide them.

Once the user selects a gadget, more information are displayed in the *Analysis View*.

5.2.2 Analysis View

(Figure 1B)

The analysis view offers an insight of the selected gadget, it shows gadget attributes and offers the user to possibility to add the gadget to the chain or to search for similar gadgets. This is done by collecting all the gadget in the tree with the same semantics, and plotting them in a graph based their different modified register when executing. This allows the user to search for gadgets with the same effect but that clobber different registers. Clicking on a displayed point will show the class of gadget in the tree view to visualize them.

The gadget placement in the scatterplot is based on Multidimensional Scaling where the dissimilarity function is simply the size of the modified registers that two gadgets have not in common.

5.2.3 Chain View

(Figure 1C)

The chain view presents the current state of the builded chain. Adding a gadget to the chain will trigger the recomputation for the registers the user has successfully set with the chain, and, by default, the view of the whole gadgets will display only the gadgets safe with respect to modified registers, that the user can choose.

It offers the possibility to move gadgets in the chain by dragging and dropping, and to remove them. Once the user is happy with the chain builded clicking `Dump` the system will generate a python script to integrate with current exploit technologies that will generate the byte-code of the chain, for example:

```
IMAGE_BASE = 0x0
rebase = lambda x : p64(x +
    IMAGE_BASE)

rop = ''
rop += rebase(0x4016ea) #pop rax;
    ret
rop += p64(0x0)
rop += rebase(0x470931) #pop rdi;
    or byte ptr [rax + 0x39], cl;
    ret
rop += p64(0x0)
rop += rebase(0x4016ea) #pop rax;
    ret
rop += p64(0x0)
rop += rebase(0x456499) #mov qword
     ptr [rdi], rax; ret
rop += rebase(0x46defd) #pop rsi;
    ret
rop += p64(0x0)
rop += rebase(0x4016ea) #pop rax;
    ret
rop += p64(0x0)
```

References

[1] Hovav Shacham, The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), CCS, 2007.

[2] Nergal, The advanced return-into-lib(c) exploits (PaX case study), Phrack Magazine 58(4), Dec. 2001.

[3] `https://github.com/sashs/Ropper`, 2018.

[4] `https://github.com/JonathanSalwan/ROPgadget`, 2017.