ROP Emporium Beginner’s Guide

How to use this site

Introduction

You already know this bit
ROP is a mechanism that can be leveraged to bypass protection schemes such as NX/DEP. For some well-meaning and only slightly misguided background on the subject you can check out the Wikipedia page. For a little more depth and some questionable sentence construction take a look at Sarif El-Sherei's ROP FTW paper.

Why you're here
ROP Emporium provides a series of challenges that are designed to teach ROP in isolation, with minimal requirement for reverse-engineering or bug-hunting. Each challenge introduces a new concept with slowly increasing complexity. Follow the links on the homepage to exercise descriptions and binaries. Sometimes a few clues as to how you might go about solving the challenge are included.

What this isn't
The focus here is on developing ROP chains to perform a desired task, everything to the left of that is bug-hunting or early exploit development and everything to the right is pentesting, neither of which are covered here. If you're new to binary exploitation consider leaving these challenges alone for now and having a crack at some of the early IO wargame levels over at smashthestack. This guide and associated challenges do not cover all existing ROP techniques, nor do they deal with mitigations.

Platforms
The ROP Emporium challenges were all compiled for Ubuntu Linux on x86 architecture.

Tools

Getting the job done
Below is a short list of tools you may find useful, it is by no means exhaustive. These tools all have their own documentation and tutorials online but some one-shot uses that will allow you to quickly find information about your target binaries are covered later in this guide.

radare2
radare2 is, amongst many other things; a disassembler, debugger and binary analysis tool. It's absurdly powerful and you have more or less everything you need to complete the challenges on this site entirely within the radare2 framework. It's actively developed and you can find more detail on their github page which also hosts a cheatsheet.

ropper
Standalone gadget finder written in Python, can also display useful information about binary files. Check out the github page for more information.

pwntools
Powerful CTF framework written in Python. Simplifies interaction with local and remote binaries which makes testing your ROP chains on a target a lot easier. Check out the github page.

General advice

Bug hunting
The ROP Emporium challenges attempt to remove as much reliance on reverse-engineering and bug-hunting as possible so you can focus on building your ROP chains. Each binary has the same vulnerability; a user-provided string is copied into a stack-based allocation with no bounds checking, allowing a function’s saved return address to be overwritten. Since this overflow occurs on the stack you can just stick your ROP chain right there and the binary will dutifully return through it. The focus is on how you design your chains around the restrictions that the challenges put in place. In some cases you may have to deal with bad characters, in others you may have a limited or obscure set of gadgets to get the job done.

32 vs 64 bits
Each challenge has 32 and 64 bit versions. The idea is to help you understand the differences in ROP chain construction between the two architectures. It's definitey worth giving both versions a try. Get familiar with both calling conventions so you know where to place your arguments before a call.

Debuggery
Once you've planned your ROP chain a debugger can make things much clearer and highlight any errors you've made early on. pwntools can launch binaries via gdb and radare2 features a powerful debugger. Debugging can give you a good indication of whether your chain will work in practice.

Automation
Using ROP chain generators such as angrop for these exercises is discouraged. Learning the concepts of ROP by manually constructing your chains is the goal. That said, don't shy away from writing your own automation tools if you find yourself always searching for the same set of gadgets in a binary, or writing some template functions for your exploits so you're not duplicating effort.

Treasure hunting

Needles in callstacks
The first issue many come across when developing their first ROP chain is gathering the information they need to begin crafting it. What useful functions are present? What selection of gadgets are available? This section contains some use cases of the tools mentioned above to help

you find this information quickly. Each challenge page gives a little more depth to what's required and in some cases more specific tool use tips as well.

Confirming protections
When approaching a challenge it's good practice to check the protections it has enabled (if any). Bear in mind that some are dependent on both the flags the binary was compiled with and the OS. The ROP Emporium challenges will all implement NX, since this is what we're trying to bypass. You can confirm you're not wasting your time by using rabin2 or checksec:

```
$ rabin2 -I <binary>
```

```
$ checksec <binary>
```

rabin2 is one of many standalone binaries that make up the radare2 suite, it will be available to you if you’ve installed radare2. checksec can be downloaded standalone from git but its functionality is also integrated into the pwntools framework which is highly recommended.

Function names
In unstripped binaries function names can sometimes be useful, especially if the programmer used helpful titles such as win_this_challenge(). You will still be able to learn the names of imported functions from stripped binaries. Listing functions imported from shared libraries is simple:

```
$ rabin2 -i <binary>
```

Listing just those functions written by the programmer is harder, a rough approximation could be:

```
$ rabin2 -qs <binary> | grep -ve imp -e ' 0 '
```

This leaves some lint but gives you something more readable than listing all symbols.

Function locations
Finding the location of a local function in an unstripped binary is as simple as listing the symbols with programs like rabin2, nm or readelf, provided the binary isn’t position independent. More often you'll want to learn the location of symbols that have been imported from shared libraries, these can be gleaned from .got.plt entries once they've been lazy-linked. These addresses can then sometimes be used to locate functions within shared libraries that haven't been linked into the binary.

Strings
Counterintuitively, don't use the 'strings' command to search for strings in the challenge binaries, use rabin2 instead. For example; running strings on the 'fluff' challenge will yield 98 lines of output to wade through, most of which are irrelevant. On the other hand, running

```
$ rabin2 -z fluff
```

will yield 5 lines, all of which are strings the programmer purposefully placed in the binary.

Common pitfalls

Here are some issues you may encounter, the causes of which aren't immediately obvious.

/usr/bin/bash
When debugging a ROP chain with gdb, if you've successfully called system() but the string you've passed is not a valid program then gdb will tell you it has started a new process "/usr/bin/bash". This can be particularly confusing when you're trying to drop a shell. Check out the system() manpage for the reason behind this.

Newlines
Some of the ROP Emporium challenges deal with avoiding badchars, but it's easy to forget that most challenges have inherent badchars due to the way they read your input. ROP Emporium challenges typically use fgets() which means you can use null bytes in your ROP chains but 0x0a will terminate your chain prematurely. Speaking of newlines; when crafting your exploits don't forget that a newline is required at the end of your string to cause the binary to process your input.

Too much
Consider the length of your chains, each binary will only read a specific number of bytes before processing your input. This is an attempt to guide players towards the desired solution to the challenge, if you're exceeding the input length for a challenge you're probably making life harder for yourself. If in doubt check that your entire chain has made it into memory intact.

Stack alignment
If you've moved the stack pointer ensure it's still correctly aligned for the architecture you're targeting. Odd things can happen otherwise.

Stack location
Think about where your chain resides as it’s being processed. If you've pivoted and placed a chain in the .data section of process memory for example then consider how much stack space is needed by a function you've called. If that function requires a lot of space it may move the stack pointer into non-writable memory or overwrite memory that will have a detrimental effect on program stability if corrupted.

Random
In some cases your exploit may not succeed for no apparent reason. Remember that you're corrupting data and sometimes these binaries will exhibit undefined behaviour. For example in the badchars challenge, a chain of particular length may cause your exploit to fail on the 32 bit binary for no apparent reason. (It’s still unclear as to why...)

Your first ROP chain

Look ma No-eXecute
Get on your way by clicking the "ret2win" challenge link on the homepage. It features a short intro on how to approach the binary, including the use of some of the tools mentioned above.