

Buffer Overflow – Practical Labs

Software Security

a.a. 2023/2024

Laurea Magistrale in Ing. Informatica

Roberto Natella



Victim program – Pearls of Wisdom

```

corso@ubuntu: ~/challenge
corso@ubuntu:~/challenge$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
Non ci sono più le mezze stagioni
Hello there
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
Sopra la panca la capra campa
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
Non ci sono più le mezze stagioni
Sopra la panca la capra campa
Hello there
1. Receive wisdom
2. Add wisdom
Selection >

```

wisdom-alt.c



Challenge

- Challenge richiesta:
 - attaccare il buffer overflow nella funzione "**put_wisdom()**" (versione **64 bit**), iniettare uno shellcode (es. hello world, oppure reverse shell)
- Challenge extra:
 - attaccare di nuovo la vulnerabilità, fare eseguire la funzione "**write_secret()**"
 - attaccare "**put_wisdom()**" nella versione a **32 bit**
 - attaccare il buffer overflow nell'array globale "**ptrs**" (versione 32 bit), fare eseguire la funzione "**pat_on_back()**"



Compilazione

- L'**eseguibile x64** (versione 64 bit) è compilato inclusivo di info di debug, con il comando

```
gcc -fno-stack-protector -z execstack -g  
wisdom-alt.c -o wisdom-alt
```

- Per compilare la **versione a x86 32 bit**:

```
gcc -fno-stack-protector -z execstack -m32 -g  
wisdom-alt.c -o wisdom-alt-32
```



Challenge - suggerimenti

- La vulnerabilità è un **classico stack overflow** su **gets()**
 - Inviare prima la stringa "2\n"
 - Poi, inserire una stringa molto lunga
- Per costruire il payload, è necessario:
 - **avere all'inizio 1024 caratteri**
("2\n" + **un riempitivo**, read() consuma 1024 caratteri)
 - a seguire, il resto del payload (es. stringa cyclic)

```
# Prima parte del payload
$ python3 -c 'import sys; sys.stdout.write("2\n" + "A"*1022)' > payload

# Seconda parte del payload (da realizzare)
$ ...TODO... >> payload
```

Challenge - suggerimenti

- Per passare il payload al programma:

```
$ ./wisdom-alt < payload
```

- Per passare il payload via GDB:

```
$ gdb ./wisdom-alt
gdb> run < payload
```



Challenge - suggerimenti

```

from pwn import *
context.arch='amd64'    # 64-bit version of x86
context.os='linux'

# Return address in little-endian format
ret_addr = # TBD: indirizzo dello shellcode (oppure di write_secret())
addr = p64(ret_addr, endian='little')

# Opcode for the NOP instruction (for NOP sled)
nop = asm('nop')

# First part of the payload
payload = b"2\n" + b"A"*1022

# Second part of the payload
payload += # TBD: riempitivo di NOP, shellcode # + addr

with open("./shellcode_payload", "wb") as f:
    f.write(payload)
  
```

Challenge extra - suggerimenti

- Per chiamare la funzione "**write_secret**", è necessario **trovare l'indirizzo delle funzione in memoria**

Step per **GDB**:

1. avviare GDB, passando il percorso del programma
2. inserire un breakpoint all'inizio del programma con "**break main**"
3. avviare il programma con "**run**"
4. ...il SO inizializza la memoria del processo...
5. l'esecuzione si interrompe subito
6. stampare l'indirizzo della funzione con "**print write_secret**"

Challenge extra - suggerimenti

```
$ gdb ./wisdom-alt
```

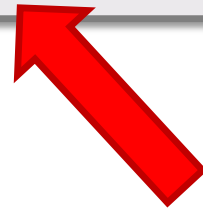
```
(gdb) break main
```

```
(gdb) run
```

...l'esecuzione si interrompe subito prima del main()...

```
(gdb) print write_secret
```

```
$1 = {void (void)} 0x55555555229 <write_secret>
```



indirizzo da usare per sovrascrivere
l'indirizzo di ritorno sullo stack



Challenge extra – versione x86-32

- L'eseguibile "**wisdom-alt-32**" è una versione a 32 bit dello stesso eseguibile
- Si implementi il porting dell'attacco
- Metterà in evidenza alcune differenze tra x86 a 64 e 32 bit

Differenze importanti:

- 1) In **x86-32**, gli indirizzi sono di **4 bytes** invece che 8. Occorre tenerne conto quando si legge lo stack in GDB, quando si utilizza pwntools, ...
- 2) Per trovare lo "offset" (la posizione nel payload in cui inserire l'indirizzo da sovrascrivere), bisogna tener conto che il comportamento della **istruzione RET in x86-32** è leggermente diverso da x86 a 64 bit (vedi prossima slide)

Challenge extra – suggerimenti (x86-32)

- In **x86-64**, la RET causa una eccezione **PRIMA di fare il pop** dell'indirizzo dallo stack
 - L'indirizzo rimane sulla cima dello stack
 - Il processore si rifiuta di scrivere in **RIP** un indirizzo "non-canonico"
- In **x86-32**, la RET causa una eccezione **DOPO aver fatto il pop**
 - L'indirizzo viene rimosso dalla cima dello stack
 - Il processore inserisce l'indirizzo in **EIP**



Challenge extra – suggerimenti (x86-32)

LA PORZIONE DI STRINGA
CICLICA DA CONSIDERARE È
SULLA CIMA DELLO STACK

```
R14 0x0
R15 0x0
RBP 0x4141414141414141 ('AAAAAAA')
RSP 0x7fffffffbb38 ← AAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAA
RIP 0x5555555531a (copier+4) ← ret
▶ 0x5555555531a <copier+49> ret <0x4141414141414141>
```

x86-64

RET viene interrotta prima di
prelevare dalla cima dello stack
(RIP non viene modificato)

LA PORZIONE DI STRINGA
CICLICA DA CONSIDERARE È
CONTENUTA IN "EIP"

```
EDI 0xf7fb0000 (_GLOBAL_OFFSET_TABLE)
ESI 0xf7fb0000 (_GLOBAL_OFFSET_TABLE)
EBP 0x41414141 ('AAAA')
ESP 0xffffad10 ← 0x41414141 ('AAAA')
EIP 0x41414141 (AAAA)
Invalid address 0x41414141
```

x86-32

RET viene interrotta dopo aver
prelevato dallo stack e copiato in EIP
(EIP viene modificato)



Challenge extra – suggerimenti (x86-32)

```

from pwn import *
context.arch='i386'      # 32-bit version of x86
context.os='linux'

# Return address in little-endian format
ret_addr = # TBD: indirizzo dello shellcode (oppure di write_secret()) #
addr = p32(ret_addr, endian='little')

# Opcode for the NOP instruction (for NOP sled)
nop = asm('nop', arch="i386")

# Writes payload on a file
payload = b"2\n" + b"A"*1022
payload += # TBD: riempitivo di NOP, shellcode # + addr

with open("./shellcode_payload", "wb") as f:
    f.write(payload)
  
```

Challenge extra – array globale

- Un'altra vulnerabilità è legata all'array globale "**ptrs**"...
 - Si provi a inserire un **valore diverso da 1 o 2!**
 - Occorre fare in modo che il programma acceda al puntatore "p" invece che ai puntatori in "ptrs"
 - Ricordare che la sintassi in C "**array[i]**" equivale a "**array + i*sizeof(array[0])**"



Challenge extra – array globale

Suggerimenti:

- 1) Prima di avviare il programma con "**run**", impostare un **breakpoint prima o dopo la read()** (es. "**break wisdom-alt.c:97**")
- 2) Determinare gli indirizzi delle variabili **buf**, **ptrs**, **p**, e delle funzioni, usando il comando "**print variabile**"
- 3) Per proseguire l'esecuzione, usare "**next**" (esegue singola istruzione, poi si ferma di nuovo) oppure "**continue**"

