Esercitazione 1

2024-25

© F. Forno, 2021-25

Obiettivi

- Capire i problemi derivanti dalla corruzione di memoria
- Gestione dello stack per le chiamate a funzione in arch x86 (AMD64)
  - organizzazione stack
  - gestione registri
  - stack contraction
- Vedere in pratica degli exploit che sfruttano corruzioni di memoria
  - Esercizi: https://exploit.education/phoenix/
  - Soluzioni passo passo: https://n1ght-w0lf.github.io/binary%20exploitation/stack-zero/

© F. Forno, 2021-25

Errori tipici nella gestione memoria

- Buffer overflow
  - un buffer fisso memorizza un input di lunghezza variabile
  - se i controlli sono fatti male dei dati possono andare oltre il buffer
  - si possono sovrascrivere altre variabili, nella stessa funzione, nel chiamante o nello heap
- Dangling e wild pointer
  - sono puntatori non inizializzati o che puntano ad aree non più in uso dello heap
  - possono rivelare dei dati precedentemente usati, ma che non dovrebbero essere più accessibili
  - si possono sovrascrivere aree arbitrarie di memoria

In genere il comportamento dopo errori di gestione memoria è imprevedibile (*per il programmatore…*)

© F. Forno, 2021-25

Conseguenze

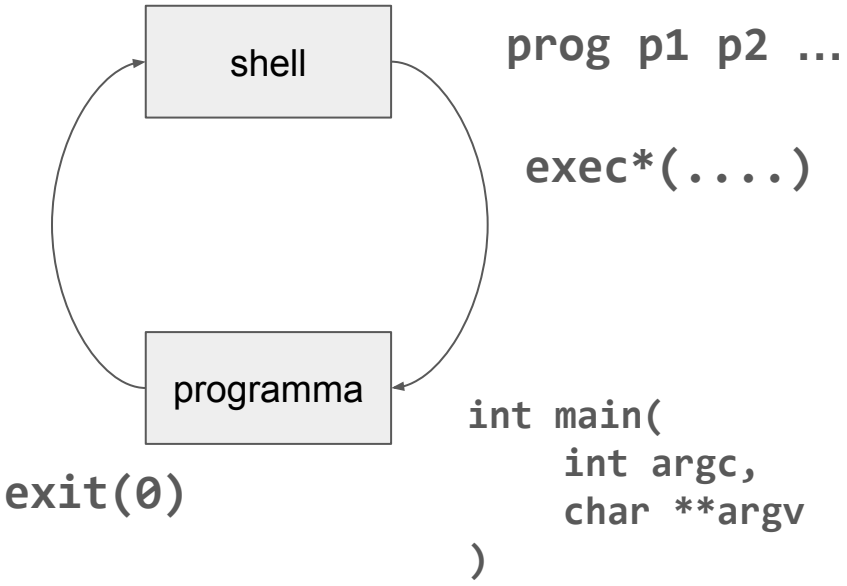
Gli errori nella gestione della memoria possono avere diverse conseguenze impreviste

- crash (molto frequente)
- lentezza (spesso nel caso di memory leak)
- cattivo funzionamento
  - es. per un problema di memoria Google non è riuscita a far convergere il calcolo del page rank per mesi intorno al 2010
- **exploit**
  - attaccanti possono sovrascrivere indirizzi e registri modificando il comportamento del programma, facendo eseguire codice arbitrario, scalare privilegi

© F. Forno, 2021-25

Invocare un programma

- Da shell si può invocare un programma passando dei parametri
  - la shell crea un nuovo processo con una system call (famiglia **exec***) che riceve i parametri e le variabili di ambiente
  - dopo lo startup nel processo viene invocato main() con i parametri, in numero e lunghezza variabile
- Variabili d'ambiente:
  - si possono leggere con la syscall **getenv()**
  -



© F. Forno, 2021-25

Analizzare codice

Come tool principali usiamo:

- gdb: debugger command line che permette di ispezionare indirizzi delle funzioni e disassemblare il codice. Es:

```
gdb -q /opt/phoenix/amd64/stack-three
gef➤  print complete_level
$1 = {<text variable, no debug info>} 0x40069d <complete_level>
disassemble main
Dump of assembler code for function main:
   0x00000000004006b5 <+0>:     push   rbp
   0x00000000004006b6 <+1>:     mov    rbp,rsp
   0x00000000004006b9 <+4>:     sub    rsp,0x60
   0x00000000004006bd <+8>:     mov    DWORD PTR [rbp-0x54],edi
   0x00000000004006c0 <+11>:    mov    QWORD PTR [rbp-0x60],rsi
```

- objdump: visualizza info sugli eseguibili, ad esempio indirizzi delle syscall chiamate.

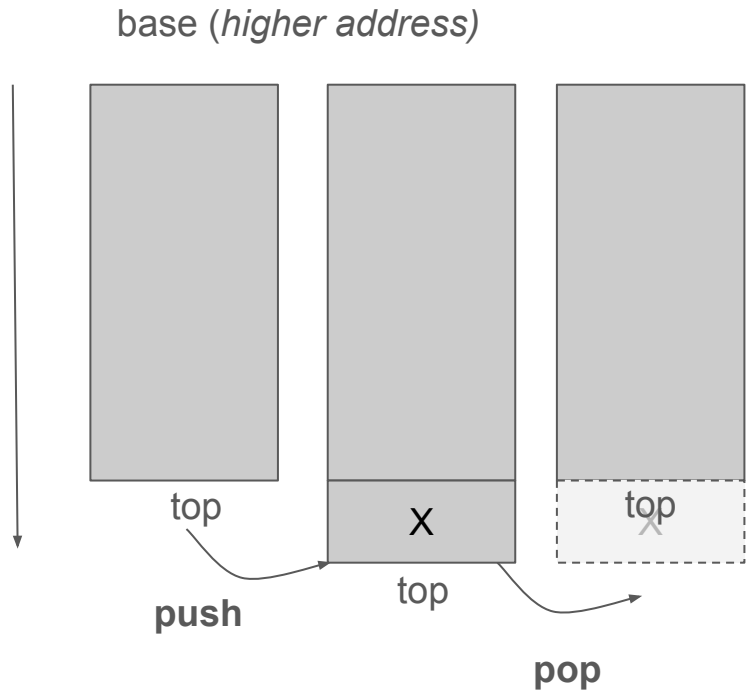
```
objdump -R /opt/phoenix/i486/stack-three
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080498a4 R_386_COPY        stdout
08049888 R_386_JUMP_SLOT   printf
0804988c R_386_JUMP_SLOT   gets
```


© F. Forno, 2021-25

Stack

Lo stack è una struttura dati che supporta le seguenti operazioni

- push X
  - X viene messo in cima allo stack
  - top = top - sizeof(X)
- pop X
  - la cima dello stack viene copiata in X
  - top = top + sizeof(X)
- top X
  - la cima viene copiata in X



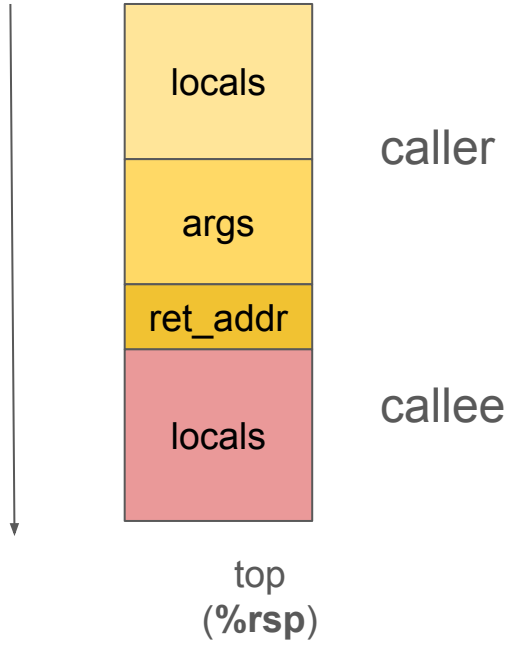
© F. Forno, 2021-25

Invocazione funzioni: uso dello stack (visione ad alto livello)

Il registro **%rsp** punta al top dello stack

Nello stack una funzione memorizza (**pushd**)

- i dati locali
  - le variabili dichiarate nello scope della funzione
  - i registri che si vuole preservare prima di invocare una funzione
- gli argomenti per il chiamato
  - in sequenza nello stack (il primo all'indirizzo più alto)
- il return address
  - l'indirizzo della prima istruzione da eseguire quando si esce dalla funzione invocata



© F. Forno, 2021-25

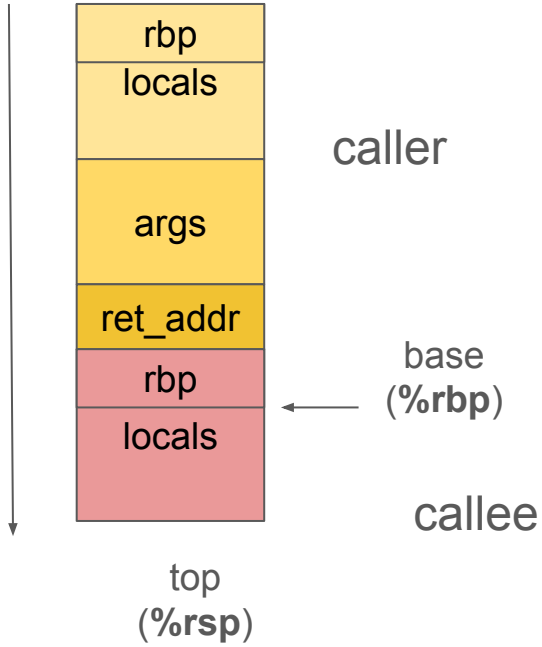
Invocazione funzioni: frame pointer (base pointer)

Tutti gli indirizzi nel proprio frame di stack sono calcolati come offset rispetto l'indirizzo iniziale del frame

L'indirizzo viene memorizzato nel registro **%rbp** (base pointer)

All'ingresso la funzione:

- salva %rbp del chiamante nello stack
- copia %rsp in %rbp che ora punta all'inizio del frame per la funzione



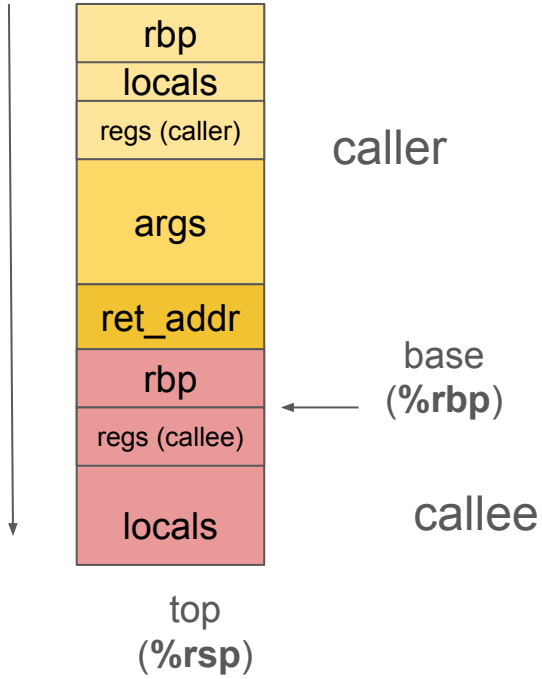
© F. Forno, 2021-25

Invocazione funzioni: salvataggio registri

I registri utilizzati dalle funzioni vanno salvati nello stack, per ripristinarne il valore all'uscita (ad esempio %rbp)

Per convenzione alcuni registri sono salvati dal chiamante prima della call, altri da chiamato.

- callee **%rbp, %rbx, %r12, %r13, %r14, and %r15**
- caller: ogni altro registro di cui vuole preservare il valore



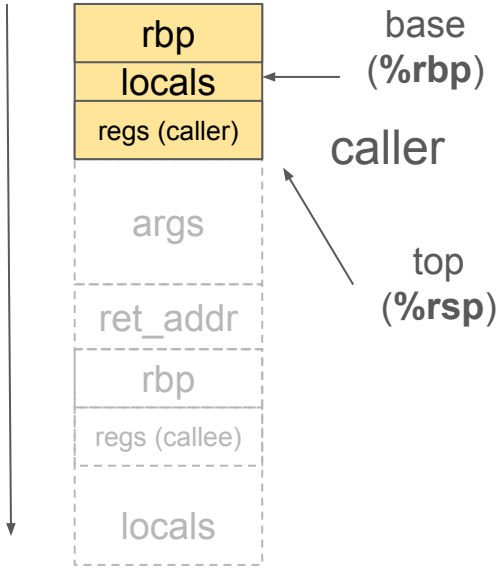
© F. Forno, 2021-25

Invocazione funzioni: ritorno

Al ritorno la funzione:

- sava il valore di ritorno in **%rax**
- ripristina tutti i registri salvati (**popd**)
- ripristina %rbp (**popd**)
- ripristina **%rip** dal saved ret_addr e ritorna il flusso al chiamante (**popd**)

L'insieme delle pop che ripristinano i valori per il chiamante viene chiamate anche "**contrazione dello stack**".
Il chiamante eventualmente ripristinerà i registri di sua competenza.



© F. Forno, 2021-25

Ottimizzazioni

In amd64 i parametri possono essere passati tramite alcuni registri dedicati

- fino a 6 parametri nei registri **%rdi, %rsi, %rdx, %rcx, %r8, %r9**
- dal settimo vanno nello stack
- parametri più piccoli di 64bit occupano comunque un registro
- parametri più grandi di 64bit possono occupare più registri
- per i float vengono usati altri registri dedicati

Questo velocizza il passaggio dei parametri in quanto si risparmiano degli accessi in ram (~100 volte più lenta)

© F. Forno, 2021-25

Shellcode

Con **shellcode** si intende del codice assembler valido (una sequenza binaria di opcode) che viene passato attraverso shell ad un programma e sfruttando qualche meccanismo di buffer overflow si cerca di farlo eseguire.

Di solito sono pochi byte, il minimo necessario per chiamare una syscall per permette di eseguire l'attacco desiderato.

Generarlo è semplice: basta compilare il programma e copiare i byte rilevanti dal file binario!

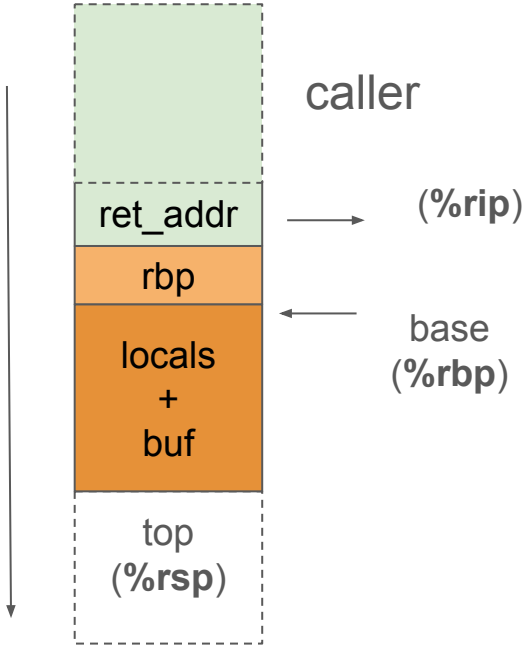
Strategia generale:

- identificare nel programma un buffer di input grande abbastanza da contenere quel codice: è una stringa, viene accettato e memorizzato!
- individuare l'indirizzo sullo stack del buffer
- provare a sovrascrivere il registro rip con l'indirizzo di quel buffer

© F. Forno, 2021-25

Shellcode in pratica

- Una funzione ha un buffer che può contenere lo shellcode e il buffer viene letto da standard input
- Se si scrive oltre il buf (gli indirizzi crescono verso l'alto) possiamo sovrascrivere rbp e il ret_addr nello stack
- Possiamo quindi inserire l'indirizzo fisico di buf in ret_addr
- Al return della funzione %rip viene ripristinato con l'indirizzo di buf e non con quello della istruzione successiva del caller
- l'esecuzione continua dallo shell code che abbiamo iniettato



© F. Forno, 2021-25

Esercizi

1. Scaricare da es01 la vm phoenix, scompattare e lanciare la vm:

```
cd exploit-education-phoenix-amd64
sh boot-exploit-education-phoenix-amd64.sh
```

2. Loggarsi con user / user (o root root)
   a. ssh -p2222 user@localhost
3. Risolvere gli esercizi seguendo il codice al link https://exploit.education/phoenix/stack-zero/ e https://n1ght-w0lf.github.io/binary%20exploitation/stack-zero/ e successivi
4. i file compilati si trovano in /opt/phoenix

© F. Forno, 2021-25