

# Software Security

Domenico Cotroneo

Roberto Natella



# GDB – basics

---

GNU Debugger, meglio noto come **gdb**, è il debugger più popolare per UNIX ed è utilizzato per effettuare il debug di programmi C e C++.

Gdb permette di ottenere un gran numero di informazioni sul programma analizzato:

- In caso di errore di segmentazione, qual è l'istruzione che lo ha causato?
- Se si verifica un errore durante l'esecuzione di una funzione, a quale riga del programma corrisponde la chiamata a funzione e quali sono i parametri?
- Qual è il valore di una variabile in un punto specifico dell'esecuzione?
- Qual è il risultato di una specifica espressione?

# GDB – comandi

---

Tra i comandi che gdb mette a disposizione ci sono:

- *run*: avvia l'esecuzione del programma
- *break n*: imposta un breakpoint alla riga *n*
- *next*: esegue l'istruzione successiva al breakpoint corrente
- *step*: analoga a *next*, con l'eccezione che in presenza di una chiamata a funzione ne esegue la prima istruzione
- *continue*: riprende l'esecuzione dopo un breakpoint
- *print var*: stampa il valore della variabile *var*

# Challenge – buffer overflows

```
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 >■
```

# Challenge – buffer overflows

---

- Quali sono le due vulnerabilità di buffer overflow nel programma?
- Come fare a forzare la chiamata di queste funzioni?
  - pat\_on\_back
  - write\_secret

# Challenge - suggerimenti

---

- Una delle vulnerabilità è legata all'array globale *"ptrs"*...
  - Si provi a inserire un valore diverso da 1 o 2!
  - Quali sono gli indirizzi delle variabili *buf*, *ptrs*, *p*, e delle funzioni?
  - Prima di avviare il programma con *"run"*, impostare un breakpoint prima o dopo la *read()* (*"break wisdom-alt.c:97"*)
  - Stampare con *"print nomevar"*, proseguire con *"next"* (singola istruzione) oppure *"continue"*
- Per sfruttare *"ptrs"*, ricordarsi che la sintassi in C *"array[i]"* equivale a *"array + i\*sizeof(array[0])"*

# Challenge - suggerimenti

---

- La seconda vulnerabilità è un classico stack overflow su `gets()`
  - Per scrivere sul buffer, serve prima inviare la stringa "2\n"
  - La prima `read()` del programma legge in totale 1024 caratteri
- Per cui, il formato del payload è più complesso, es.:

```
$ python -c 'import sys; sys.stdout.write("2\n"+"A"*1022)'> payload_search
```

```
$ cat pattern_payload >> payload_search
```
- Per analizzare il contenuto dello stack durante l'attacco
  - usare ancora breakpoint ("*break wisdom-alt.c:63*")
  - avanzare con "*next*" e "*stepi*" (avanza di una singola istruzione assembly, per analizzare l'uso dello stack durante l'istruzione C di "*return*")