

Progetto Internet-Security

Claudio Nuncibello

appello 10 luglio 2023

Sommario

1	Introduzione ai buffer overflow	2
1.1	Attacchi di tipo buffer overflow	2
1.2	Prime possibili misure di sicurezza	4
1.2.1	Permessi e memoria	4
2	Attacchi con riuso del codice	4
2.1	exploit; riuso del codice	5
2.2	ultime parole sul riuso del codice	8
2.3	mitigazione exploit 1	8
2.3.1	Attenzione al codice	8
2.3.2	Canarini	9
2.3.3	Casualizzazione dello spazio degli indirizzi	9
3	Attacchi senza diversione del flusso del controllo	10
4	Conclusione	10
5	Sitografia e bibliografia	12

Ai fini della trattazione sarà utile esaminare la natura dei *buffer overflow*. Da sempre sono considerati, a ragione, tra le più fastidiose vulnerabilità di un software; banalmente perchè, nonostante siano ben noti agli sviluppatori, rimangono alte le probabilità di errore nel codice, versioni software non aggiornate o bug non noti le cui occorrenze sono inaspettate.

1 Introduzione ai buffer overflow

Cosa intendiamo quando ci riferiamo ad un *overflow del buffer*? il buffer è una memoria temporanea usata per memorizzare comandi di input e output; quando qualcosa nella logica del programma va storto è possibile inserire più dati di quelli che il buffer può contenere e così si verifica un *overflow del buffer*. Uno dei motivi per cui questi tipi di attacchi sono così ricorrenti è dovuta al fatto che la stragrande maggioranza dei sistemi operativi e la maggior parte dei programmi di sistema sono scritti in linguaggio C/C++, sfortunatamente nessun compilatore C/C++ esegue controllo dei limiti degli *array*.

1.1 Attacchi di tipo buffer overflow

In questa trattazione ci concentreremo sugli attacchi allo *stack*. Abbiamo descritto cos'è un *overflow di un buffer*, ma come avviene e cosa succede all'interno della memoria del programma? Per capirlo dobbiamo dare uno sguardo a come il sistema operativo "spinge" i dati nello *stack*; prendiamo in esempio un server in c che, esemplificando, se inserita la password corretta "droppa" una *shell* all'utente.

```
int main(int argc, char **argv)
{
    printf("WELCOME TO THE SECURE SERVER");

    if (checkPassword())
    {
        debug();
    }
}
```

Figure 1: main

Guardiamo questo pezzo di codice derivante dal main: La funzione *checkPassword()* valida la password inserita dall'utente. Quando il programma salta alla funzione *checkPassword()* spinge per prima cosa sullo *stack* l'indirizzo di ritorno, cioè l'indirizzo del *main()*.



Figure 2: funzione gets

Successivamente, la variabile locale, di grandezza 64 byte, viene spinta in cima allo *stack*. La funzione *gets* non fa altro che leggere una stringa di dimensione ignota e la inserisce dentro il nostro buffer di dimensione fissa. Il problema sta nella mancanza di controllo sulla dimensione dell'input da parte della funzione *gets*.

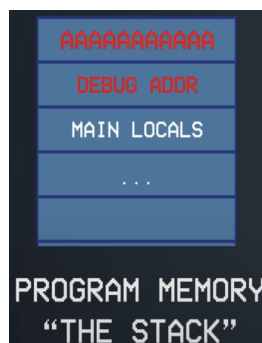


Figure 3: overflow del buffer

Cosa succede quindi se la stringa inserita dall'utente è troppo grande? La *Figure 3* mostra la situazione, come si è visto la funzione *gets* copia tutti i byte nel buffer e anche al di là, sovrascrivendo molti dati presenti nello *stack* ma soprattutto sovrascrivendo l'indirizzo di ritorno che vi era posto; in altre parole una parte della stringa password ora riempie una locazione di memoria che, per

il sistema, dovrebbe contenere l'indirizzo dell'istruzione a cui passare quando la funzione esegue *return*.

1.2 Prime possibili misure di sicurezza

1.2.1 Permessi e memoria

Supponiamo ora che ad inserire la stringa troppo lunga non sia un utente innocente, ma un attaccante che inserisce un messaggio appositamente costruito per bypassare il flusso di controllo del programma; per esempio l'attaccante potrebbe inserire una stringa preparata in modo da sovrascrivere l'indirizzo di ritorno con l'indirizzo del buffer. Il risultato è che quando il programma salta all'indirizzo che lui crede "legittimo", in verità torna all'inizio del buffer inserito dall'attaccante, eseguendo come codice i byte che contiene. La vera causa del problema quindi non è che l'aggressore riesca a sovrascrivere i puntatori a funzione e gli indirizzi di ritorno, ma che riesca ad iniettare del codice e che questo sia eseguito. A questo punto verrebbe da chiedersi: non si può fare in modo da rendere impossibile l'esecuzione di byte sull'*heap* e sullo *stack*? Buona idea! gli **attacchi di tipo code injection** non funzionano più se i byte forniti dall'aggressore non possono essere eseguiti come codice legittimo. Le moderne CPU hanno una funzione nota come **bit NX**, che sta per "*No-eXecute*", ossia divieto di esecuzione. È molto utile distinguere fra segmenti dati (*heap*, *stack* e variabili globali) e segmenti testo (che contengono il codice); in particolare, molti sistemi operativi moderni cercano di fare in modo che i segmenti di dati siano scrivibili ma non eseguibili e che quelli di testo siano eseguibili ma non scrivibili. Per verificare e settare questa funzionalità in *SO linux* possiamo usare questo comando :

```
parallels@debian-gnu-linux-11:~$ sudo sysctl -w kernel.exec-shield=2
```

Questa politica è nota in *OpenBSD* come **W XOR X** e significa che la memoria può essere o eseguibile o scrivibile, ma non entrambe le cose. *MacOS*, *Linux* e *Windows* hanno sistemi di protezione simili. Un nome generico per questa misura di sicurezza è *DEP* (Data Execution Prevention, divieto di esecuzione dei dati). Alcuni hardware non supportano il *bit NX*; in questi casi e comunque possibile far funzionare la *DEP*, ma l'implementazione sarà meno efficiente. La *DEP* impedisce tutti gli attacchi che usano la logica descritta dall'esempio sopra, l'aggressore può iniettare tutto lo *shell-code* che vuole in un processo, ma a meno che sia in grado di rendere eseguibile la memoria non ci sarà modo di eseguirlo.

2 Attacchi con riuso del codice

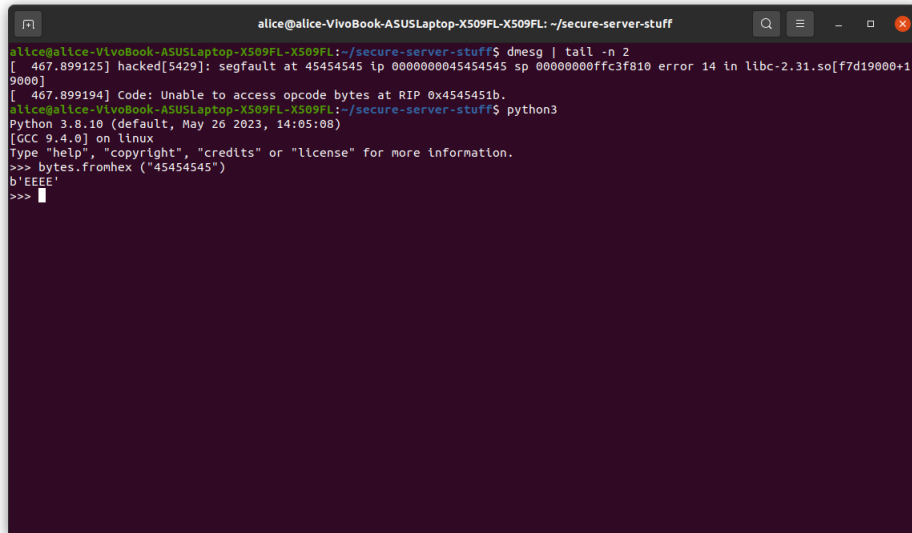
La *DEP*, dunque, impedisce l'esecuzione di codice in una regione di dati, purtroppo però, i nostri problemi non finiscono qui, perché c'è chi ha pensato ad altri at-

2.1 exploit; riuso del codice

```
alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL: ~/secure-server-stuff
alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL:~/secure-server-stuff$ ./hacked
WELCOME TO THE SECURE SERVER
password: password
Wrong password, sorry;
alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL:~/secure-server-stuff$ ./hacked
WELCOME TO THE SECURE SERVER
password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCCDDDEEEFFFF
Errore di segmentazione (core dump creato)
alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL:~/secure-server-stuff$
```

In questa demo noi rappresenteremo gli attaccanti. Vediamo l'esecuzione del server, che ironicamente chiameremo *hacked*. Il server funziona in questo modo: ci richiede una password la quale, se errata blocca il processo, mentre se corretta ci fornisce la shell per eseguire i nostri comandi. Nel nostro scenario ipotizziamo che la shell *droppata* dal server non sia la *shell* dell'utente ma una con altre caratteristiche e permessi funzionali per il server *hacked*. L'attacco prosegue con l'inserimento dell'input malevolo, diamo per scontato che abbiamo già scoperto che il buffer in questione abbia una lunghezza fissa di 64 byte, per tanto capiamo, andando avanti con l'esempio il motivo dietro la scelta degli

ultimi caratteri della stringa.

A terminal window with a dark purple background. The title bar reads 'alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL: ~/secure-server-stuff'. The terminal output shows a dmesg command being run, followed by a segmentation fault message from the kernel. Then, a Python 3.8.10 prompt is shown, followed by a help message and a bytes.fromhex call. The prompt is currently at the end of the last line.

```
alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL: ~/secure-server-stuff$ dmesg | tail -n 2
[ 467.899125] hacked[5429]: segfault at 45454545 ip 0000000045454545 sp 00000000ffc3f810 error 14 in libc-2.31.so[f7d19000+19000]
[ 467.899194] Code: Unable to access opcode bytes at RIP 0x4545451b.
alice@alice-VivoBook-ASUSLaptop-X509FL-X509FL: ~/secure-server-stuff$ python3
Python 3.8.10 (default, May 26 2023, 14:05:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> bytes.fromhex ('45454545')
b'EEEE'
>>>
```

Figure 5: Segmentation fault

dmesg è un comando tipico dei sistemi operativi Unix e Unix-like che visualizza ,sullo standard output, i messaggi contenuti nel buffer del kernel del sistema operativo, combinato al comando *tail* visualizziamo a schermo l'errore di Segmentation fault e nello specifico l'indirizzo a cui stava cercando di saltare il programma. A questo punto non ci resta che capire quali sono i byte del nostro input che hanno sovrascritto l'indirizzo di ritorno, così da sostituirli a nostro piacimento. Ma con cosa dobbiamo sostituire i byte interessati?

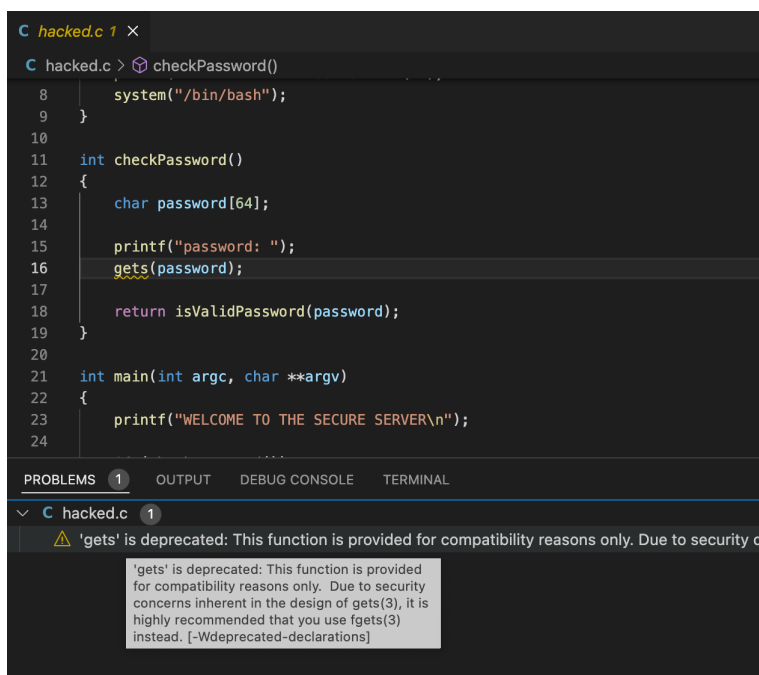
2.2 ultime parole sul riuso del codice

Abbiamo visto come invece di eseguire uno *shell-code* scritto dall'attaccante, quest'ultimo può semplicemente riscrivere lo *stack* con una stringa contenente l'indirizzo di una funzione già esistente. Il trucco della **programmazione orientata al ritorno (ROP, Return-Oriented Programming)** è quindi cercare brevi sequenze di codice che (a) facciano qualcosa di utile e (b) terminino con un'istruzione `return`. L'aggressore può concatenare queste sequenze utilizzando l'indirizzo di ritorno che ha passato allo *stack*; i singoli brani di codice prendono il nome di *gadget* e hanno di solito una funzionalità estremamente limitata, come sommare due registri, caricare un valore dalla memoria in un registro o passare un valore allo *stack*. In altre parole, l'insieme dei *gadget* può essere considerato come una sorta di strano set di istruzioni utilizzato dall'aggressore per realizzare funzionalità arbitrarie, manipolando lo *stack* in maniera intelligente.

2.3 mitigazione exploit 1

di seguito vediamo delle accortezze di programmazione e delle tecniche utili a diminuire l'esposizione dei nostri programmi a questo tipo di attacchi

2.3.1 Attenzione al codice



```
C hacked.c 1 x
C hacked.c > checkPassword()
8   system("/bin/bash");
9   }
10
11  int checkPassword()
12  {
13      char password[64];
14
15      printf("password: ");
16      gets(password);
17
18      return isValidPassword(password);
19  }
20
21  int main(int argc, char **argv)
22  {
23      printf("WELCOME TO THE SECURE SERVER\n");
24  }
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

▼ C hacked.c 1

⚠ 'gets' is deprecated: This function is provided for compatibility reasons only. Due to security concerns inherent in the design of gets(3), it is highly recommended that you use fgets(3) instead. [-Wdeprecated-declarations]

'gets' is deprecated: This function is provided for compatibility reasons only. Due to security concerns inherent in the design of gets(3), it is highly recommended that you use fgets(3) instead. [-Wdeprecated-declarations]

L'esempio visto prima non funziona solamente per i programmi che usano *gets* (anche se bisognerebbe proprio cercare di evitare di usare questa funzione),

ma per qualsiasi codice che copia in un buffer dati forniti da un utente senza controllare le violazioni dei limiti. I dati dell'utente possono essere costituiti da parametri della riga di comando, stringhe di ambiente, dati inviati su una connessione di rete o dati letti da un file. Esistono numerose funzioni che copiano o spostano questi dati: *strcpy*, *memcpy*, *strcat* e molte altre. Ovviamente, qualsiasi ciclo scriviate che sposta byte in un buffer può essere altrettanto vulnerabile. Compilatori molto diffusi quali *Visual Studio*, *gcc* e *LLVM/Clang* offrono dei "disinfettanti" come opzione in fase di compilazione per bloccare un'ampia gamma di possibili attacchi. Uno dei più noti è Address Sanitizer. Compilando il codice con `-fsanitize-address`, il compilatore si accerta che tutte le allocazioni di memoria siano affiancate da zone rosse, cioè piccole aree di memoria "non valida". Qualunque accesso a una zona rossa, ad esempio derivante da un *buffer overflow*, porta a un crash del programma accompagnato da un messaggio d'errore opportunamente deprimente. Per farlo, AddressSanitizer mantiene una mappa di bit per indicare che ciascun bit della memoria allocata è valido, mentre ciascun bit della zona rossa è non valido. Ogni volta che il programma accede alla memoria, controlla rapidamente la mappa per vedere se l'accesso è consentito. Naturalmente tutto ciò non è gratis: la mappa di bit e le zone rosse aumentano il consumo di memoria, e l'inizializzazione e la consultazione della mappa penalizzano molto le prestazioni. È raro che ai product manager piaccia un fattore quasi 2 di rallentamento, quindi AddressSanitizer è poco usato nel codice di produzione. È invece molto utile in fase di test. In nostro aiuto vengono gli editor di testo più moderni che hanno parecchi flag che avvertono il programmatore dell'utilizzo di funzioni pericolose.

2.3.2 Canarini

il nome viene dall'utilizzo che ne facevano i minatori in miniera, quando il canarino moriva voleva dire che l'aria si stava riempiendo di gas tossici ed era quindi l'ora di risalire in superficie. I moderni sistemi informatici utilizzano canarini (digitali) come sistemi di allarme precoce: nei punti in cui un programma fa una chiamata di sistema, il compilatore inserisce del codice che salva sullo *stack*, subito sotto l'indirizzo di ritorno, un valore casuale che funge da canarino. Al rientro da una funzione, il compilatore verifica il valore del canarino; se è cambiato significa che qualcosa è andato storto. In questo caso, meglio mandare in crash il programma che continuare l'esecuzione.

2.3.3 Casualizzazione dello spazio degli indirizzi

Ecco un'altra idea per bloccare questi attacchi. Oltre a modificare l'indirizzo di ritorno e a iniettare alcuni programmi o pezzi di codice già esistenti, l'aggressore dev'essere in grado di ritornare esattamente all'indirizzo giusto: La cosa è semplice se gli indirizzi sono fissi, ma se non lo sono? La **ASLR** (Address Space Layout Randomization) cerca di rendere casuale l'indirizzo delle funzioni e dei dati a ogni esecuzione del programma; in questo modo per l'attaccante è molto più difficile attuare un exploit del sistema. L'**ASLR**, in particolare, può casu-

alizzare le posizioni dello *stack* iniziale, dell'*heap* e delle librerie. Molti moderni sistemi operativi supportano, oltre ai *canarini* e la **DEP**, anche la **ASLR** sia per il sistema operativo che per le applicazioni utente, anche se con casualità ("entropia") diversa. La forza combinata di questi tre meccanismi di protezione ha alzato di parecchio l'asticella per gli attaccanti: anche solo saltare al codice iniettato o a qualche funzione in memoria è diventato un lavoraccio. Insieme, questi tre meccanismi sono un'importante linea di difesa dei moderni sistemi operativi; il bello è che offrono protezione a un costo assolutamente accettabile per le prestazioni. Per verificare e settare questa contromisura è possibile usare il comando:

```
parallels@debian-gnu-linux-11:~$ sudo sysctl -w kernel.randomize_va_space=2
```

3 Attacchi senza diversione del flusso del controllo

Fino ad adesso abbiamo visto come sfruttare a nostro vantaggio il trabocco di dati di un *buffer* per modificare indirizzi di ritorno o puntatori a funzioni. Lo scopo era sempre lo stesso: fare in modo che il programma esegua pezzi di codice non previsti e con priorità elevata. Questa però non è l'unica possibilità, anche i dati possono essere un bersaglio seducente per un attaccante. La certezza che abbiamo è che il gioco attacco/difesa sarà in continuo moto; in tutto il mondo i ricercatori stanno studiando nuove possibilità di difesa; alcune di queste hanno come obiettivo i file binari, altre consistono in estensioni di sicurezza ai compilatori C e C++, come Address Sanitizer visto prima. È importante sottolineare che anche gli attaccanti stanno affinando le loro tecniche. Abbiamo cercato di dare un'occhiata ad alcune delle più importanti, ma esistono parecchie varianti di una stessa idea. La buona notizia è che le cose stanno migliorando, Molti di questi exploit sono dovuti al fatto che C e C++ sono linguaggi molto permissivi e con pochi controlli, che rendono molto veloci i programmi. Linguaggi più moderni, come Rust e Go, sono molto più sicuri; certamente i programmi non sono veloci quanto quelli in C o in C++, ma oggi (più che 30 o 40 anni fa) si tende ad accettare una prestazione meno brillante se i programmi hanno meno difetti.

4 Conclusione

Al fine di mantenere i nostri sistemi più sicuri, capiamo che non esiste una misura di sicurezza o di prevenzione che ci salvaguardi in assoluto da ogni tipo di attacco. La maniera migliore per difendersi è spesso un insieme di tecniche e accortezze che creano una sicurezza a più livelli utile al nostro scopo. Tra di queste, a parere di chi scrive, la più importante che l'utente possa implementare è l'aggiornamento software poiché, come abbiamo visto, *kernel* più recenti usano

tecniche sempre più moderne e anche la sola riorganizzazione della memoria nel nostro caso specifico potrebbe giocare a nostro vantaggio, dato che software vecchi sono i più esposti allo studio dagli attaccanti, che saranno in grado di utilizzare le loro tecniche in modo più intelligente ed efficace.

5 Sitografia e bibliografia

1. [CWE-121-Overflow del buffer basato su stack](#)
2. Git hub server in c : <https://github.com/lowlevellearning/secure-server-stuff/>
3. Andrew S. Tanenbaum, Herbert Bos, I moderni sistemi operativi edizione 2023.
4. medium.com profilo di lowlevellearning <https://medium.com/@lowlevellearning/your-first-buffer-overflow-b44e5ba5598a>
5. Oracle linux kernel guide <https://docs.oracle.com/en/operating-systems/oracle-linux/6/security/>