Exploittare vulnerabilità delle format string **by styx^**

0 Pre introduzione

Salve sono styx^,questa è l'unico testo **completo** in italiano sulle vulnerabilità delle format string e su come exploittarle(o almeno credo). Questo articolo non è di mia creazione,ma di quei simpaticoni della TESO:io ho pensato solamente a tradurlo.Il perché è molto semplice:non conoscendo molto bene l'inglese per capire questo testo l'unico modo era tradurlo e poi rileggermelo. Ed eccolo qua!
Una volta tradotto ho deciso di renderlo pubblico,perché ho pensato di fare un gran favore a quelle persone che come me conoscono poco l'inglese oppure solamente non gli va di tradurre testi. Per tradurre questo testo –anche se non sembra vero- ci ho messo molte ore:diciamo più di cinque. Lo sforzo comunque mi è stato ripagato ed ora ne so qualcosa di queste vulnerabilità. Quando leggerete questo testo dovete tener presente che:
1)Ho 4 in italiano
2)alcune parole ho deciso di lasciarle in inglese poiché in italiano non rendono(es. brute forcing)
3)alcune parole me le sono inventate(es. exploittare e simili)
4)quello dei TESO scrive da cani
5)ci sono alcune espressioni mai sentite,ma grazie all'aiuto al mio fratellino tradotte(es. to trigger)
Bene, partendo da questo presupposto potete iniziare a leggere questo articolo maledetto. Per prendermi in giro sulla traduzione,per contatti, per complimenti o simili potete contattarmi ad ouchls@tin.it o su IRC(irc.azzurra.it) a:
#ondaquadra
#spine
#hacklab
#oltrelinux
#slackware
#programmazione
#C
#crack- it
Buona lettura!

Exploiting Format String Vulnerabilities

scut / team teso
September 1, 2001
version 1.2

1 Introduzione

Questo articolo spiega il fenomeno della natura del fenomeno che ha scioccato la comunità della sicurezza nella seconda metà dell'anno 2000. Conosciuta come 'la vulnerabilità delle format strings,un nuova intera classe di vulnerabilità è stata scoperta e causò che furono scoperti un'ondata di bug exploittabili in tutti i tipi di programma,che andavano dalle piccolo utilities alle applicazioni dei grandi server. Questo articolo proverà a spiegarti la struttura della vulnerabilità a dopo con l'uso di queste conoscenze la costruzione di sofisticati exploit. Ti mostrerà come scoprire vulnerabilità nel codice C,e perché questo nuovo tipo di vulnerabilità è più pericoloso che la comune vulnerabilità del buffer overflow.
L'articolo è basato su un discorso tedesco Che ho dato al diciassettesimo Chaos Communication
Congress [2] a Berlin, Germania. Dopo il discorso ho ricevuto numerose richieste per tradurlo e ho ricevuto numerosi commenti positive. Tutto questo mi ha motivato di rivisitare il documento,updatarlo e correggere i dettagli a fare una versione LaTEX più

utilizzabile.
Questo articolo copre la maggior parte delle cose menzionate negli altri articoli,più alcuni trucchi per l'exploitation. Una volta letto mandatemi commenti,idee o ogni altra cosa non scurrile all'indirizzo scut@team- teso.net .
La prima parte dell'articolo tratta della storia e i particolari delle vulnerabilità del format string,seguito da dettagli su come scoprire ed evitare vulnerabilità nel source code. Poi sono sviluppate le tecniche di base,dalle quali sorgono metodi di exploitation più potenti. Questo metodo è poi modificato,migliorato e applicato praticamente su speciali situazioni che ti permettono di exploittare quasi ogni tipo di vulnerabilità delle format string viste fino adesso.
Come con ogni vulnerabilità è stata sviluppata nei vecchi tempi,e nuove tecniche sono mostrate,spesso perché le vecchie non funzionano in certe situazioni. Le persone,che meritano veramente credito per molte tecniche menzionate in questo articolo e hanno influenzato il mio scrivere significamene sono tf8,che scrisse il primo exploit per le format string,portal,che sviluppò a ricercò exploittabilità nel suo eccellente articolo[3], DiGiT,che trovò la maggior parte delle vulnerabilità critiche remote conosciute oggi,e smiler,che sviluppo tecniche di brute force sofisticate.
Sebbene ho anche contribuito per qualche trucchetto,senza il aiuto,commenti e trucchi, - entrambi teoricamente o in forma di exploit – mostratimi da queste persone,questo articolo non sarebbe stato possibile.
Versione corrette e aggiornate potrebbero apparire nell'homepage del TESO Security Group[1].

1.1 Buffer Overflows vs. Format String Vulnerabilities

Poichè quasi tutte le vulnerabilità del passato erano tipi di buffer overflows,uno potrebbe paragonare tale vulnerabilità seria e di basso livello rispet to a questo nuovo tipo di vulnerabilità.

Buffer Overflow Format String

public since
danger realized
number of
exploits
considered
techniques
visibility
mid 1980's
1990's
a few thousand
as security threat
evolved and advanced
sometimes very difficult to
spot
June 1999
June 2000
a few dozen
programming bug
basic techniques
easy to find

1.2 Statistiche: importanti vulnerabilità format string nel 2000

Per sottolineare l'impatto pericoloso che le vulnerabilità delle format string hanno avuto nell'anno 2000,abbiamo esposto la vulnerabilità più exploittate pubblicizzate.
Application Found by Impact Years
wu- ftpd 2.*

Linux rpc.statd
IRIX telnetd
Qualcomm Popper 2.53
Apache + PHP3
NLS / locale
screen
BSD chpass
OpenBSD fstat
security.is
security.is
LSD
security.is
security.is
CORE SDI
Jouko Pynn¯onen
TESO
ktwo
remote root
remote root
remote root
remote user
remote user
local root
local root
local root
local root
> 6
> 4
> 8
> 3
> 2
?
> 5
?
?

Ci sono ancora un sacco di vulnerabilità sconosciute e non scoperte lasciate al tempo di scrivere,e per i prossimi due o tre anni le vulnerabilità delle format string contribuiranno alle statiche delle nuove vulnerabilità che sono trovate. Come potrai vedere,sono molto facili da scoprire automaticamente con molti tools sofisticati,e puoi assumere che per la maggior parte delle vulnerabilità nei codici d'oggi che non sono ancora conosciuti pubblicamente,un exploit già esiste.

2 The format functions

La format function è uno speciale tipo di funzione dell' ANSI C,che prende un numero variabile di argomenti,dai quali uno è chiamato così formato stringa. Mentre la funzione valuta il formato stringa,essa accede ai parametri extra dati alla funzione. E' una funzione conversione,che è usata per rappresentare i data types primitivi del C in una rappresentazione di stringa leggibile dall'uomo. Esse sono usate in ogni programma C,per fare l'output di informazioni,scrivere messaggi di errore o stringhe di processo.
In questo capitolo noi tratteremo le tipiche vulnerabilità nell'uso delle funzioni format,il corretto uso,alcuni dei loro parametri a il generale concetto del "format string vulnerability".

2.1 Come compaiono le format string vulnerabili?

Se un attacker può stabilire la format string in un funzione di format dell'ANSI C in parti o come completa,la "format string vulnerability" è presente.Così facendo,il comportamento di una funzione format è cambiato,e l'attacker potrebbe avere il controllo sopra l'applicazione bersaglio.
Nell'esempio sottostante,la stringa dell'user è fornita dall'attacker—egli può controllare l'intera ASCIIZ- string,per esempio attraverso l'uso di un parametro di linea di comando.
2.2 La famiglia delle funzioni di format:
Uso sbagliato:

```
Int func (char *user)
{
printf (user);
}
```

Corretto:

```
Int func (char *user)
{
printf ("%s", user);
}
```

2.2 La famiglia delle format function
Un numero di format function sono definite nella definizione ANSI C.Ci sono alcune fondamentali funzioni stringa sui quali sono basate funzione più complesse,alcune delle quali non fanno parte dello standard,ma sono ampiamente disponibili.
I membri della famiglia reale:
• fprintf — scrive su un FILE stream
• printf — scrive sull 'stdout' stream
• sprintf — scrive dentro una stringa
• snprintf — scrive dentro una stringa con la verifica della lunghezza
• vfprintf — scrive in un FILE stream da una struttura va_arg
• vprintf — scrive sull 'stdout' stream da una struttura va_arg
• vsprintf — scrive dentro una stringa da una struttura va_arg
• vsnprintf — scrive dentro una stringa con la verifica della lunghezza da una struttura va_arg
Relativi:
• setproctitle — set argv[]
• syslog — output to the syslog facility
• altri come err*, verr*, warn*, vwarn*

6.2 LE FUNZIONI DI FORMAT

2.3 L'uso delle funzioni di format

Per capire dove questa vulnerabilità è solito in un codice C,dobbiamo esaminare lo scopo delle format functions.
Funzionalità:
• usata per convertire dei semplici datatypes C in una rappresentazione di stringa
• permette di specificare il formato della rappresentazione
• elaborare la stringa risultante (output to stderr, stdout, syslog, ...)
Come funzionano le format function:
• la format string controlla il comportamento della funzione
• specifica il tipo di parametri che dovranno essere scritti
• I parametri sono salvati sullo stack (pushed)
• entrambi sono salvati direttamente (per valore), o indirettamente (per riferimento)
La funzione chiamante:
• deve conoscere quanti parametri sono pushati nello stack, poichè effettua una connessione con lo stack, quando ritorna la funzione

2.4 Cosa è esattamente una format string?

Una format string è una stringa ASCIIZ che contiene testo e parametri format.
Esempio:
printf("Il numero magico è: %d\n", 1911);
Il testo da essere scritto è "Il numero magico è:",seguito da un parametro format '%d', che è sostituito dal parametro (1911) nell'output.Dunque l'output apparirà come: Il numero magico è: 1911.
Alcuni parametri format:
Il parametro output passato come:
%d valore decimale (int)
%u valore decimale senza segno (unsigned int)
%x valore esadecimale (unsigned int)
%s riferimento a stringa ((const)(unsigned)char*)
%n riferimento al numero di byte fin qui (*int)
Il carattere '\' è usato per i speciali caratteri escare.E' sostituito dal compilatore C al momento della compilazione,sostituendo la sequenza escape con l'appropriato carattere nel binario. Le funzione di format non riconoscono queste speciali sequenza. Infatti,non hanno niente a che vedere con le funzioni di format,ma sono qualche volta mischiati,come se sono valutate da esse.
Esempio:
printf("Il numero magico è: \x25d\n", 23);
Questo codice funziona,perché '\x25' è sostituito al momento della compilazione con '%',poiché 0x25 (37) è il valore ASCII per il carattere di percentuale.
2.5 Lo stack e le sue regole alle format string
Il comportamento delle funzioni di format è controllato dalla format string. Le funzioni recuperano i parameri richiesti dalla format string dallo stack.
printf("Il numero %d non ha indirizzo,il numero %d ha: %08x\n",i,a,&a);
Dall'interno della funzione printf lo stack appare come:
cima dello stack
…
<&a>
<a>
<i>
A
…
fondo dello stack
dove:
A indirizzo della format string
i valore della variabile i
a valore della variabile a
&a indirizzo della variabile i
La funzione di format ora analizza la stringa di format 'A',dalla lettura di un carattere per volta. Se esso non è '%',il carattere è copiato nell'output. Nel caso in cui lo sia,il carattere che segue '%'specifica il tipo di parametro che dovrebbe essere valutato. La stringa "%%" ha un significato speciale,è usata per scrivere lo stesso carattere di escare '%'. Ogni altro parametro collegato con i dati,il quale è localizzato nello stack.

3 Format string vulnerabilities

La classe generica della vulnerabilità del format string è un 'problema di incanalamento'. Questo tipo di vulnerabilità può apparire se due differenti tipi di canali di informazione sono uniti in uno solo, e caratteri speciali o sequenze di escape sono usate per distinguere quale canale è correntemente attivo. Il più delle volte un canale è un canale dati,il quale non è analizzato attivamente ma solamente copiato,mentre l'altro canale è il canale di controllo.

Mentre questa non una cosa cattiva in sé,può diventare velocemente un terribile problema di sicurezza se un attacker è capace di fornire l'input che è usato in un canale. Spesso ci sono delle routine di escape o di de- escape difettose, o sono dirette a un livello,proprio come nelle vulnerabilità del format string. Così per farla corta:I problemi di incanalamento non hanno buchi di sicurezza in se,ma essi creano dei bug exploittabili.
Per illustrare questo problema generale,c'è una tavola di più comuni problemi di incanalamento:
Situation Datachannel Cont rolling channel Security
problem
Phone systems Voice or data Cont rol tones seize line control
PPP Protocol Transfer data PPP command s traffic amplification
Stack Stack data Return addresses cont rol of retaddr
Malloc Buffers Malloc data Management info write to memory
Format strings Output string Format parameter s format function
cont rol
Tornando alla specifica vulnerabilità del format string,ci sono due tipiche situazioni,dove le vulnerabilità del format string possono presentarsi:
- Tipo uno (come su Linux rpc.statd,IRIX telnetd).Qui la vulnerabilità giace nel secondo parametro della funzione syslog. La stringa di format è parzialmente fornita dall'utente.

```
char tmpbuf[512];
snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```

- Tipo due (come in wu- ftpd, Qualcomm Popper QPOP 2.53). Qui una stringa fornita dall'utente è parzialmente o completamente passata indirettamente a una funzione format.

```
int Error (char *fmt, ...);
...
int someotherfunc (char *user)
{
...
Error (user);
...
}
...
```

Mentre le vulnerabilità del primo tipo sono sicuramente scoperti da un tool (come pscan o TESOgcc),vulnerabilità del secondo tipo possono essere individuate solo se è detto al tool che la funzione 'Error' è usata come una funzione di format.

3.1 Cosa controlliamo ora?

Tuttavia fornendo la stringa di format noi si in grado di controllare il comportamento della funzione di format. Dobbiamo ora esaminare cosa esattamente siamo in grado di controllare,e come usare questo controllo per espandere questo controllo parziale sopra il processo per il pieno controllo del flusso d'esecuzione.

3.2 Crash del programma

Un semplice attacco usando la vulnerabilità del format string è di crashare il processo. Questo può essere utile per alcune cose,per esempio crashare un processo che scarica il core e ci potrebbero essere dati utili dentro il coredump. O in alcuni network attacks è utile avere un servizio che non risponde,per esempio con il DNS spoofing.
Comunque,ci potrebbe essere qualche interesse nel crashare un processo. Quasi tutti i punti di accesso illegali nei sistemi UNIX sono presi dal kernel e il processo manderà un

segnale SIGSEV. Il programma termina normalmente e scarica il core.
Dall'utilizzo delle format strings possiamo facilmente scovare qualche punto invalido di acceso solo dal fornire una stringa come:
printf ("%s%s%s%s%s%s%s%s%s%s%s");
Poichè '%s' mostra la memoria da un indirizzo che è fornito nello stack,dove anche molti altri dati giacciono,le nostre possibilità sono alte di leggere da un indirizzo illegale,il quale non è indicato. Anche molte implementazioni della format function offrono il parametro '%n',il quale può essere usato per scrivere verso un indirizzo sullo stack. Se questo è fatto per poche volte,dovrebbe anche produrre un crash attendibile.

3.3 Vedendo il processo di memoria

Se noi possiamo vedere la risposta della format function –la stringa di output - ,noi possiamo raccogliere informazioni utili da essa,poichè è l'output del comportamento che noi controlliamo,e possiamo utilizzare questi risultati per guadagnare una vista generale di cosa fa la nostra format string e come il disegno del processo appare. Questo può essere utile per varie cose,come trovare il corretto offset per la reale exploitation o solo per ricostruire lo stack frames del processo bersaglio.

3.3.1 Vedendo lo stack

Noi possiamo mostrare alcune parti della memoria dello stack usando una stringa come questa:
printf ("%08x.%08x.%08x.%08x.%08x\n");
Questa funziona poichè noi induciamo la funzione printf a ricevere cinque parametri dallo stack e a mostrarli come numeri esadecimali . Così un possibile output può essere:
40012980.080628c4.bffff7a4.00000005.08059c04
Questo è un dump parziale della memoria di stack,partendo dal fondo corrente fino alla cima dello stack. A seconda della grandezza del buffer del format string e dalla grandezza dell'output buffer,tu puoi ricostruire più o meno grandi parti della memoria di stack con l'utilizzo di questa tecnica. In alcuni casi tu puoi comunque ricevere l'intera memoria dello stack.
Il dump dello stack da importanti informazioni circa il flusso del programma e le variabili della funzione locale e può essere di grande aiuto per trovare il corretto offset per una exploitation trionfante.

3.3.2 Vedendo la memoria a qualunque locazione

E' anche possibile sbirciare alle locazioni di memoria differenti dalla memoria dello stack. Per fare questo noi dobbiamo ottenere la funzione di format per mostrare la memoria da un indirizzo che forniamo noi. Questo pone due problemi:Primo,noi dobbiamo trovare un parametro di format che usa un indirizzo(per riferimento) come parametro stack e mostrare la memoria da qui,e dobbiamo fornire quell'indirizzo. Siamo fortunati nel primo caso,poiché solo il parametro '%s' fa quello,mostra la memoria –solitamente in una stringa ASCIIZ- da un indirizzo fornito dallo stack. Così il problema rimanente è come ottenere quell'indirizzo nello stack,dentro il giusto posto.

3.4 Sovrascrittura di memoria arbitraria

La nostra stringa è solitamente è solitamente situata nello stack lei stessa,così già siamo vicini al pieno controllo sopra lo spazio,dove la format string giace.
La funzione di format internamente mantiene un puntatore alla locazione del corrente parametro di format. Se saremmo in grado di ottenere questo puntatore puntando ad uno spazio di memoria che noi possiamo controllare,possiamo fornire un indirizzo al parametro '%s'. Per modificare il puntatore allo stack possiamo utilizzare uno stupido parametro che scaverà in su lo stack stampando cianfrusaglie:
printf ("AAA0AAA1_%08x.%08x.%08x.%08x.%08x");
Il parametro '%08x' incrementa il puntatore interno dello stack della funzione di format verso la cima dello stack. Dopo più o meno l'incremento di questi parametri il puntatore punta nella nostra memoria:la stessa format string. La funzione di format mantiene

sempre il più basso frame dello stack,così il nostro buffer giace su nello stack in tutto,giace si sicuro sopra il puntatore corrente dello stack. Se noi scegliamo il numero di parametri '%08x' correttamente,noi potremmo mostrare la memoria da un indirizzo arbitrario,con l0aggiunta di '%s' nella nostra stringa. Nel nostro caso l'indirizzo è illegale e dovrebbe essere 'AAA0'. Lasciamo rimetterlo a posto da uno reale.
Esempio:
address = 0x08480110
address (tradotto come come una stringa a 32 bit): "\x10\x01 \ x48\ x08"
printf ("\x10\x01\ x48 \ x08_%08x.%08x.%08x.%08x.%08x|%s|");
Scaricherà la memoria da 0x08480110 fino ad arrivare al NULL byte. Dall'incremento dell'indirizzo della memoria dinamicamente noi possiamo tracciare fuori l'intero spazio del processo. E'anche possibile creare un coredump come immagine di un processo remoto e ricostruire un binario da quello[13]. E' anche di molto aiuto per trovare la causa di una tentata exploitation senza successo.
Se noi non possiamo arrivare all'esatto confine della format string dall'uso di un pop di 4 byte ('%08x'),dobbiamo percorrere la format string,aggiungere uno,due o tre caratteri cianfrusaglia. Questo è analogo all'allineamento negli exploit del buffer overflow.

3.4 Sovrascrittura di memoria arbitraria

Il santo graal di una exploitation è prendere il controllo del puntatore di istruzioni di un processo. Nel maggiore dei casi il puntatore di istruzione (spesso chiamato IP o PC) è un registro nella PCU e non può essere modificato direttamente,poiché solo le istruzioni macchina possono cambiarlo. Ma se siamo in grado di pubblicare queste istruzioni macchina noi abbiamo bisogno di avere il controllo. Così noi non possiamo direttamente prendere il controllo sopra il processo. Normalmente il processo ha più privilegi che di quelli di un attacker.
Invece dobbiamo trovare le istruzioni che modificano il puntatore di istruzioni e capire come queste istruzioni lo modificano. Questo sembra complicato,ma nel maggiore dei casi è molto facile,poiché ci sono istruzioni che prendono un puntatore di istruzioni dalla memoria e saltano su di esso. Così nella maggior parte dei casi controlla sopra questa parte di memoria,dove un puntatore di istruzioni è situato,è il precessore a controllare lo stesso puntatore di istruzioni. Questo è come funzionano la maggior parte dei buffer overflow.
In un processo di due stage,prima un puntatore di istruzioni salvato è sovrascritto e quando il programma esegue una legittima istruzione che trasferisce il controllo a un indirizzo fornito dall'attacker.
Noi esamineremo due differenti modi per effettuare queste vulnerabilità della format string.

3.4.1 Exploitation – simile ai comuni buffer overflows

Le vulnerabilità delle format string qualche volta offrono la via attorno alle limitazioni della lunghezza del buffer e permettono l'exploitation che è simile a quella dei comuni buffer overflow.
Un codice come questo qui sotto appare nel QPOP 2.53 e nel bftpd:

```
{
char outbuf[512];
char buffer[512];
sprintf (buffer, "ERR Wrong command: %400s", user);
sprintf (outbuf, buffer);
}
```

Casi simili sono spesso nascosti profondamente nella vita reale del codice e non ovvi come mostrato nell'esempio sopra. Fornendo una stringa speciale siamo in grado di raggirare la limitazione del '%400s':
"%497d\x3c\xd3\ x f f \ xbf <nops><shellcode>"

Ogni cosa è simile a una normale stringa del buffer overflow,solo l'inizio - "%947d" – è differente. In un normale buffer overflow noi sovrascriviamo l'indirizzo di ritorno (RET) del frame di una funzione sullo stack. Come la funzione che ha il possiede il suo frame di ritorno,ritorna l'indirizzo fornito da noi. L'indirizzo punta da qualche parte all'interno dello spazio "<nop>". Ci sono buoni articoli che descrivono questo metodo di exploitation e se questo esempio non è chiaro,ti conviene leggere prima un articolo che lo spieghi[5].
Crea una string ache contiene 497 caratteri. Insieme alla stringa di errore ("Err Wrong command: ") questa eccede del buffer 'outbuf' di 4 byte. Sebbene è permesso alla stringa 'user' di essere lunga 400 bytes,noi possiamo estendere la sua lunghezza abusando dei parametri della format string. Poiché il secondo sprintf non verifica la lunghezza,essa può essere usata per oltrepassare i confini di outbuf. Mentre ogni parametro format che permette di allungare l'originale format string,come fanno "% 50d","%50f" o "%50s",è meglio scegliere un parametro che referenzia un puntatore o potrebbe causare una divisione per zero. Queste regole escludono "%f" a "%s". Noi siamo lasciati con i parametri output di interi:"%d","%u" e "%x".
La libreria GNU C contiene un bug,che risulta in un crash se utilizzi un parametro come "%nd" con 'n' più grande di 1000. Questo è l'unico modo per determinare l'esistenza di una libreria GNU C in modo remoto. Se usi "%.n" esso lavora correttamente,eccetto se sono usati valori molto alti. Per un articolo più profondo per la lunghezza di '%nd' e '%. nd' vedi l'articolo[3].
3.4.2 Exploitation - attraverso format strings pure
Se non possiamo applicare in nessun modo la semplice exploitation appena menzionata,possiamo ancora exploittare il processo. Così facendo noi estendiamo il nostro molto limitato controllo –l'abilità di controllare il comportamento della stringa format – al reale controllo di esecuzione,che sta eseguendo il nostro grezzo codice macchina. Guarda il codice come quello trovato nel wu- ftpd 2.6.0:

```
{
char buffer[512];
snprintf (buffer, sizeof (buffer), user);
buffer[sizeof (buffer) - 1] = '\0';
}
```

Nel codice qui sopra non è possibile allargare il nostro buffer con l'inserimento di qualche tipo di allungamento di parametro,perché il programma usa la funzione sicura snprintf per assicurarsi che non possiamo eccedere il buffer.
A primo impatto sembra che non possiamo fare cose molto utili,eccetto che crashare il programma ed esaminare qualche memoria.
Bisogna ricordare il parametro menzionato. C'è il parametro '%n',che scrive il numero di byte già scritti,in una variabile scelta da noi. L'indirizzo della variabile è data alla funzione format posizionando un puntatore intero dentro lo stack.

```
int i;
printf ("foobar%n\n", (int *) &i);
printf ("i = %d\n", i);
```

Dovrebbe scrivere i=6. Con lo stesso metodo usato sopra per scrivere memoria da indirizzi arbitrari,possiamo scrivere in una locazione arbitraria:

```
"AAA0_%08x.%08x.%08x.%08x.%08x.%n"
```

Con il parametro '%08x' noi aumentiamo il puntatore interno dello stack della funzione format di 4 byte. Facciamo così finchè questo puntatore punta all'inizio della format string ('AAA0'). Questo funziona,perché normalmente la nostra format string è situata nello stack,in cima del frame della normale format function. Il '%n' scrive all'indirizzo 0x30414141,che è rappresentato dalla stringa 'AAA0'. Normalmente questo dovrebbe crashare il programma,poiché questo indirizzo non è tracciato. Ma se forniamo un corretto indirizzo tracciato e scrivibile questo lavora e noi sovrascriviamo 4 bytes (sizeof

(int)) all'indirizzo:
int a;
printf ("%10u%n", 7350, &a);
/* a == 10 */
int a;
printf ("%150u%n", 7350, &a);
/* a == 150 */
Usando uno stupido parametro '%nu' noi siamo in grado di controllare il contatore scritto da'%n',al minimo un bit. Ma per scrivere numeri – come gli indirizzi – questo non è sufficiente,così dobbiamo trovare il modo per scrivere dati arbitrari.
Così un numero come 0x0000014c è posizionato in memoria come
""\x4c\x01 \ x00 \ x00". Per il contatore nella funzione format possiamo controllare il meno importante byte,il primo byte situato nella memoria dall'uso dello stupido parametro '%nd' per modificarlo.
Esempio:
unsigned char foo[4];
printf ("%64u%n", 7350, (int *) foo);
uando ritorna la funzione printf,foo[0] contiene '\x40',che è l'equivalente di 64,il numero che utilizziamo per incrementare il contatore.
Ma per un indirizzo,ci sono 4 bytes che dobbiamo controllare completamente.Se noi non siamo in grado di scrivere 4 bytes per volta,possiamo provare a scrivere un byte per volta per 4 volte in una fila. Nella maggior parte dell'architettura CISC è possibile scrivere in un indirizzo arbitrario non allineato. Questo può essere utilizzato per scrivere al secondo meno significante byte della memoria,dove l'indirizzo è situato. Questo appare come:
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
/* 0 * before */ strcpy (canary, "AAAA");
/* 1 */ printf ("%16u%n", 7350, (int *) &foo[0]);
/* 2 */ printf ("%32u%n", 7350, (int *) &foo[1]);
/* 3 */ printf ("%64u%n", 7350, (int *) &foo[2]);
/* 4 */ printf ("%128u%n", 7350, (int *) &foo[3]);
/* 5 * after */ printf ("%02x%02x%02x%02x\n", foo[0], foo[1],foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02x\n", canary[0],canary[1], canary[2], canary[3]);
Ritorna l'output 10204080" e "canary: 00000041". Sovrascriviamo 4 volte il meno significante byte di un intero a cui puntiamo. Dall'incremento del contatore ogni volta,il byte meno significante si muove attraverso la memoria su cui vogliamo scrivere,e ci permette di immagazzinare dati arbitrari.
Come puoi vedere nella prima fila della figura 1,tutti gli 8 bytes non sono toccati ancora dal codice che sovrascrive. Dalla seconda file attiviamo 4 sovrascritture,spostate di un byte sulla destra per ogni step. L'ultima fila mostra lo stato finale desiderato:abbiamo soprascritto tutti i 4 bytes del nostro array foo,ma mentre facciamo così,noi distruggiamo 3 bytes dell'array canary. Includiamo l'array canary solo per vedere che noi sovrascriviamo memoria che non vogliamo sovrascrivere.
Figura 1:Il 4 passi per sovrascrivere un indirizzo:
1 **00 00 00 00 41 41 41 41**
00
prim
a
2 **10 00 00 00 41 41 41 41**
00
3 **10 20 00 00 00 41 41 41**
00
4 **10 20 40 00 00 00 41 41**

00
5 **10 20 40 80 00 00 00 41**
00
6 **10 20 40 80 00 00 00 41**
00
dopo
Sebbene questo metodo appare complesso,può essere usato per sovrascrivere dati arbitrari ad indirizzi arbitrari. Per spiegare abbiamo usato solo una scrittura per il format string,ma è anche possibile scrivere più volte dentro una format string:
strcpy (canary, "AAAA");
printf ("%16u%n%16u%n%32u%n%64u%n",1, (int *) &foo[0], 1, (int *) &foo[1],1, (int *) &foo[2], 1, (int *) &foo[3]);
printf ("%02x%02x%02x%02x\n", foo[0], foo[1],foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02x\n", canary[0],canary[1], canary[2], canary[3]);
Noi usiamo il parametro uno come argomento stupido per il riempimento del nostro '%u'. Anche il riempimento è cambiato,poiché il contatore dei caratteri è già a 16 quando noi vogliamo scrivere 32. Così noi dobbiamo solo aggiungere 16 caratteri al posto di 32 ad esso,per avere il risultato che vogliamo.
Questo era uno speciale caso,nel quale tutti i byte incrementavano quelli scritti. Ma potremmo anche scrivere 80 40 20 10 con solo una modifica minore. Poiché scriviamo numeri interi,solo il byte meno significante è importante negli scritti. Usando il contatore dei caratteri 0x80,0x140,0x220 e 0x310 rispettivamente quando "%n" è attivato,noi possiamo costruire la stringa desiderata. Il codice per calcolare il calcolatore desiderato numero di caratteri scritti è questo:
write_byte += 0x100;
already_written %= 0x100;
padding = (write_byte - already_written) % 0x100;
if (padding < 10)
padding += 0x100;
dove 'write_byte' è il byte che vogliamo creare,'already_written' è il contatore corrente dei bytes scritti che la funzione format mantiene e 'padding' è è il numero di bytes che dobbiamo incrementare con il contatore.
Esempio:
write_byte = 0x7f;
already_written = 30;
write_byte += 0x100; /* write_byte is 0x17f now */
already_written %= 0x100; /* already_written is 30 */
/* afterwards padding is 97 (= 0x61) */
padding = (write_byte - already_written) % 0x100;
if (padding < 10)
padding += 0x100;
Ora la format string di '%97' aumenta il contatore '%n' così che il byte meno significante equivale a 'write_byte'. Il controllo finale se il padding è minore di 10 riserva di qualche attenzione. Un semplice output di un intero,come "%u" può generare una stringa di una lunghezza superiore di 10 caratteri,dipendendo dal numero che manda in output. Se la lunghezza necessaria è più grande del padding che specifichiamo,dice che vogliamo mandare in output '1000' con un "%2u",nostro valore cadrà per non perdere nessun output significante. Dalla garanzia che il nostro padding è sempre più grande di 10,possiamo tenere sempre un numero preciso di 'already_written',il contatore mantiene la funzione format,poiché noi scriviamo sempre esattamente tanti bytes in output quanti specificati nella opzione di lunghezza nel parametro del format.
La sola cosa rimanente per exploittare tale vulnerabilità in una via pratica è di mettere gli argomenti nel giusto ordine sullo stack ed utilizzare il pop una sequenza di stackpop

per incrementare il puntatore allo stack. Esso dovrebbe essere:
A
<stackpop><dummy - addr - pair * 4><write- code>
Stackpop: La sequenza di popping di parametri dello stack che incrementa il puntatore stack. Una volta che lo stackpop è stato attuato,il puntatore interno dello stack della funzione format punta all'inizio delle stringhe dummy- addr - pair.
Dummy- addr - pair: Quattro paia di valori interi stupidi e gli indirizzi dove scrivere. Gli indirizzi sono incrementati di uno ogni paia,il valore dell'intero stupido può essere ogni cosa che contiene NULL byte.
Write- code: La parte della stringa format che attualmente fa la scrittura alla memoria,usando il paio '%nu%n',dove n è maggiore di 10.La prima parte è usata per incrementare o fare un overflow del meno significante byte della funzione format interna al contatore di byte scritti,e il '%n' è usata per scrivere questo contatore agli indirizzi che sono dentro la parte dummy- addr - pair della stringa.
Il codice scrittura deve essere modificato per eguagliare il numero di bytes scritti dallo stackpop,poiché lo stackpop scrisse già caratteri all'output quando la funzione format analizza il codice scrittura – il contatore della funzione format non inizia da zero e questo deve essere considerato.
L' indirizzo che stiamo per scrivere è chiamato Return Address Location,short retloc,l'indirizzo che creiamo con la nostra stringa di format a questo posto è chiamata Return Address,short redaddr.

4 Variazioni dell'exploitation

L'exploitation è un arte. Come in ogni arte c'è più di una via per compiere le cose. Spesso non vuoi andare sulla buona via passata di exploittare cose,ma prende vantaggio del tuo bersaglio sviluppo,sperimentando,scoprendo e usando comportamenti nel programma. Questo sforzo extra può ripagarti in molte cose,la prima essendo la sicurezza e la robustezza del tuo exploit. O se solo una piattaforma o sistema è affetto da una vulnerabilità puoi prendere vantaggio delle speciali caratteristiche del sistema per trovare uno shortcut da exploittare. Ci sono un sacco di cose che possono essere usate,questa è solo una panoramica di base delle tecniche comuni.

4.1 Scrittura breve

Invece di scrivere per quattro volte è anche possibile sovrascrivere un indirizzo con sole due operazioni di scrittura. Questo è possibile attraverso la normale operazione '%n' a la stringa '%nu' con larghi valori 'n'. Ma per questo caso speciale possiamo prendere vantaggio di una speciale operazione di scrittura che scrive che scrive short int types:il parametro '%hn'. L''h' può essere utilizzato negli altri parametri format anche,per perdere il valore fornito nello stack ad uno short type. La tecnica di scrittura breve ha un vantaggio rispetto alla prima tecnica:non distrugge i dati dietro l'indirizzo,così se ci sono validi dati dietro un indirizzo che stai sovrascrivendo,come un parametro funzione,viene preservato.
Ma in generale tu dovresti evitare ciò,sebbene questo è supportato dalla maggior parte delle librerie C:è dipendente dal comportamento della funzione format,cioè se il contatore interno dei caratteri interni può superare il confine del buffer.
Questo non lavora sulle vecchie librerie GNU C (libc5). Esso consuma anche più memoria nel processo bersaglio.
printf ("%.29010u%hn%.32010u%hn",1, (short int *) &foo[0],1, (short int *) &foo[2]);
Questo è utile specialmente per I RISC based system,che hanno restrizioni dell'allineamento per la direttiva '%n'. Usando il qualificatore short,l'allineamento è emulato nel software o sono usate speciali istruzioni macchina,a puoi solitamente scrivere su ogni confine di due byte.
Inoltre quello lavora esattamente come nella tecnica dei 4 byte. Alcune persone dicono persino che è possibile fare la scrittura in un colpo,usando specialmente larghi

riempimenti come "'%.3221219073u". Ma la pratica mostra che ciò non funziona nella maggior parte dei sistemi. Una un'analisi più profonda di questo tema lo trovi nell'articolo [3]. Altre buone informazioni le trovi sul paper di HERT[4].

4.2 Stack Popping

Un problema può sorgere se la format string è troppo breve per fornire una sequenza dello stack poppino che acceda alla nostra stringa Questa è una gara tra la reale distanza alla tua format string e la grandezza della format string,nel quale tu devi poppare al minimo la reale distanza. Così c'è una domanda per un metodo effettivo per incrementare il puntatore allo stack con meno byte possibili. Correntemente abbiamo usato solo la sequenza '%u',per mostrare il principio,ma ci sono metodi più efficaci. Una sequenza di '%u' è lunga 2 bytes e poppa 4 bytes,che da un rapporto di 1:2 (noi utilizziamo 1 byte per ricevere in cambio 2 bytes).
Attraverso l'uso del parametro '%f' noi riceviamo in cambio perfino 8 bytes nello stack,mentre utilizziamo solamente 2 bytes. Ma questo ha un enorme svantaggio,poiché se il rifiuto dallo stack è scritto come un numero float,potrebbe esserci una divisione per zero,che crasherebbe il sistema. Per evitare questo possiamo usare uno special qualificatore format,che scrive solamente la parte intera del numero float: '%.f' camminerà in salita nello stack di 8 bytes,usando solo tre bytes del nostro buffer.
Sotto i derivati BSD e IRIX,è possibile abusare del qualificatore '*' per i nostri scopi. E' usato per fornire dinamicamente la lunghezza dell'output che un parametro format produrrà. Mentre '%10d' scrive 10 caratteri, '%*d' recupera la lunghezza dell'output dinamicamente: il prossimo parametro format nello stack glielo fornisce. Poiché la LibC's menzionata sopra permette parametri del tipo '%*******d',possiamo tirare 4 bytes per ogni '*',che mette riporta al rapporto 4 a 1. Ciò crea ancora un altro problema:non possiamo predire la lunghezza dell'output nella maggior parte dei casi,poiché è settata dinamicamente dal contenuto dello stack.
Ma possiamo ignorare la specificazione dinamica emettendo un valore difficilmente codificato dietro a tutti gli starti: '%********10d' scriverà sempre 10 bytes,senza nessun problema di cosa era sbirciato dallo stack prima. Questo trucco fu scoperto da lorian.

4.3 Direct Parameter Access

Dietro al perfezionamento del metodo dello stack popping,c'è un'enorme semplificazione che è conosciuta come 'direct parameter access',un modo per indirizzare direttamente un parametro stack da dentro la format string. Quasi tutte le librerie C in uso supportano questa features,ma non tutte sono utilizzabili per applicare questo metodo per una exploitation della format string.
Il direct parameter access è controllato dal qualificatore '$':

```
printf ("%6$d\n", 6, 5, 4, 3, 2, 1);
```

scrive '1' perché '$6' esplicitamente indirizza al sesto parametro nello stack. Usando questo metodo l'intera sequenza dello stack pop può essere tirata fuori.

```
char foo[4];
printf ("%1$16u%2$n""%1$16u%3$n""%1$32u%4$n""%1$64u%5$n",1,(int *) &foo[0], (int *) &foo[1],(int *) &foo[2], (int *) &foo[3]);
```

Creerà "\x10\x20 \ x40\ x80" in foo. Questo accesso diretto è limitato a solo I primi otto parametric nei derivati BSD,eccetto IRIX. La libreria C Solaris lo limita a i primi 30 parametri.
Se scegli valori negativo o enorme destinandoli all'accesso dei parametri dello stack sotto la tua corrente posizione non produrrà il risultato aspettato ma un crash.
Sebbene semplifica molto l'exploitation,dovresti utilizzare la tecnica dello stackpop ogni volta che è possibile,poiché rende il tuo exploit più portabile. Se il bug esiste solamente su una piattaforma che permette questo metodo,potrai sicuramente prendere vantaggio di questo(per esempio il demone telnet exploittato da LSD fa questo[21]).

5 Brute Forcing

Quando exploitti una vulnerabilità come il buffer overflow o una vulnerabilità del format string spesso fallisci perché non è stato tenuto conto dell'ultimo ostacolo,di ricevere tutti i giusti offsets. Fondamentalmente trovare i giusti offsets vuol dire 'cosa scrivere e dove scrivere'. Per semplici vulnerabilità tu puoi attendibilmente indovinare il corretto offsets,oppure giusto un brute force,tentandoli uno dopo l'altro. Ma tanto presto quanto hai bisogno bisogno di multipli offsets questo problema cresce esponenzialmente,diventa impossibile un brute force.
Con le format string questo problema appare solo stai exploittando un demone o ogni programma che ti offrirà solo un tentativo. Quanto presto quanto tu hai tentavi multipli o puoi vedere il responso di una format string,questo è possibile,sebbene non futile per trovare tutti gli offsets necessari.
Questo è possibile perché abbiamo già un controllo limitato sopra il processo bersaglio che vogliamo prendere completamente:la nostra format string già dirige il processo remoto su cosa fare,permettendoci di sbirciare nella memoria o testare certi comportamenti.
Poiché i due metodi spiegati sono molto differenti,sono spiegati separatamente.

5.1 Responso Basato sul Brute Force

Prendere vantaggio di vedere la risposta del format scritto fu per primo rilevato nella più popolare format string dell'exploit per wu- ftpd 2.6.0 da tf8. Usò la risposta per determinare la distanza.
Inoltre smiler e me stesso sviluppammo questa tecnica per determinare i due altri indirizzi,l'indirizzo di ritorno 'retaddr' e la locazione dell'indirizzo di ritorno 'retloc' e lo usammo per costruire completamente un offset indipendente per l'exploit del wu- ftpd (7350wu [22]).
Per brutare(to bruteJ) la distanza dovresti utilizzare una stringa come questa:
"AAAABBBB|stackpop|%08x|"
Lo stackpop dipende dalla distanza che vogliamo indovinare. La distanza è incrementata per ogni tentativo:

```
while (distance > 0) {
strcat (stackpop, "%u");
distance - = 4;
}
```

Se vogliamo esplorare una distanza di 32,la format string apparirebbe come:
"AAAABBBB|%u%u%u%u%u%u%u%u|%08x|"
Noi poppiamo 32 bytes dallo stack (8 * "%u") e scriviamo i 4 bytes al trentaduesimo byte dallo stack esadecimale. Nel caso ideale l'output apparirebbe come:
AAAABBBB|98321793817763956176013460872891302 1|41414141|
'41414141' è la rappresentazione esadecimale di 'AAAA',abbiamo colpito l'esatta distanza di 32 bytes. Se non puoi accedere a questa tipo dall'incremento della distanza può avere due ragioni:entrambi la distanza è troppo grande per essere accessa,per esempio se la format string è localizzata nell'heap,o l'allineamento non è su un confine di 4 bytes. Nel caso più recente,dobbiamo solamente trasformare la format string da uno a più caratteri stupidi. Poi possiamo far scivolare la posizione della stringa,così che il pattern '42414141' diventa il pattern corretto '41414141'.
Una volta che possiedi l'allineamento e la distanza puoi iniziare a fare un brute force degli indirizzi del buffer della format string. Dunque usi una format string tipo questa:
addr|stackpop|_____%%|%s|
La format string è elaborate da sinistra a destra,l' 'addr' e la sequenza '__' non fa nulla di nocivo. Lo 'stackpop' muove il puntatore dello stack in alto finchè non punta all'indirizzo 'addr'. Finalmente,'%s' ora scrive la stringa ASCIIZ da 'addr'.
Nel caso ideale 'addr' potrebbe puntare nella sequenza '__' della nostra format string. In questo caso l'output apparirebbe come:

garbage|_____%|_____%%|%s||
Dove ‘garbage’ inteso come fuori dall’output ‘addr’ e ‘stackpop’. Allora la stringa elaborata ‘___%%’ è convertita in ‘___%’,come ‘%%’ è convertito in ‘%’ dal processo di format. Quando la stringa ‘_____%%|%s|’ è inserita come il ‘%s’ della nostra format string fornita,è elaborato. Nota che c’è una sola cosa che varia come noi proviamo valori differenti per ‘addr’. Nel nostro caso ideale abbiamo usato un ‘addr’ che punta direttamente nello stack. Come puoi vedere,attraverso la visione del ‘%%’ possiamo distinguere tra gli indirizzi che puntano dentro la format string(quelli che hanno due caratteri ‘%’) e l’indirizzo che punta casualmente dentro il buffer bersaglio(che ha solo un carattere ‘%’,poiché era stato convertito dalla funzione di format).
Se ‘addr’ punta dentro l’indirizzo bersaglio,l’output appare come:
garbage|_____%|_____%||
Come puoi vedere,è visibile solo un ‘%’. Questo ci permette di predire esattamente l’indirizzo del buffer bersaglio,che può essere utile per situazioni dove buffer della format string è nello stesso heap.
Poiché noi sappiamo dove è locato il nostro ‘%s’ relativamente all’inizio dello nostra format string,e abbiamo l’indirizzo che punta nel nostro buffer,noi possiamo relocalizzare l’indirizzo così che sappiamo esattamente a quale indirizzo inizia la nostra format string. Poiché usualmente vuoi mettere anche il tuo shellcode dentro la format string,puoi esattamente calcolare il relativo retaddr all’indirizzo della format string.

5.2 Blind Brute Forcing

Il brute forcing cieco non è tanto lontano quanto lo è la risposta basata sul brute forcing. L’idea di base è che possiamo misurare il tempo preso per il computer remoto per analizzare la format string. Una stringa come “%.9999999u” ne prende di più rispetto a un semplice ‘%u’. Possiamo anche produrre attendibilmente un segmentation fault usando “%n” su un indirizzo non tracciato.
L’approccio di base per questo tipo di brute force fu inventato fa tf8,dopo perfezionato da me stesso per fare un brute force agli indirizzi del buffer.
Poiché questo attacco è relativamente complesso e solo utile per certe situazioni,ho fornito un esempio lavorante nella directory example. E’ interessante se puoi azionare la vulnerabilità più volte,ma tu non vedi la risposta della funzione format,per esempio nel servio syslogs. Se sei interessato a questa tecnica,guarda i sources,mi dispiace ma ho omesso la descrizione qui.

6 Casi Speciali

Non ci sono situazioni certe dove puoi prendere vantaggio delle situazioni,non devi conoscere tutti gli offsets o puoi fare una semplice exploitation,,più ordinato,preciso e più importante:attendibile. Ho sentito pochi approcci comuni per exploittare una vulnerabilità delle format string.

6.1 Bersagli alternativi

Influenzati attraverso la lunga storia del buffer overflows basato sullo stack,un sacco di persone pensano che sovrascrivere l’indirizzo di ritorno situato nello stack è l’unico modo per avere controllo sul processo. Ma se exploittiamo una vulnerabilità format string non sappiamo esattamente dove è il nostro buffer w ci sono cose alternative che possiamo sovrascrivere. Il comune buffer overflow basato sullo stack permette di sovrascrivere solo l’indirizzo di ritorno,perché quelli sono situati anche nello stack. Comunque con le funzioni format,possiamo scrivere dovunque nella memoria,permettendoci di modificare l’intero spazio scrivibile del processo.
E’ dunque interessante da esaminare altri modi per avere il parziale o pieno controllo del processo exploittato. In alcune situazioni questo può risultare una via facile per l’exploitation o –come puoi vedere bene- può essere usato per circoscrivere certe protezioni.
Discuterò le locazioni dell’indirizzo alternativo qui brevemente,dando riferimento ad

articolo più approfonditi.

6.1.1 GOT overwrite

La spazio del processo di ogni binario ELF[12] include una speciale sezione,chiamata 'Global Offset Table'(GOT). Ogni funzione di libreria usata dal programma ha un'entry che contiene un indirizzo dove è localizzata la reale funzione. Questo è fatto per permettere una facile relocalizzazione delle librerie dentro la memoria del processo invece di usare un indirizzo di difficile codifica. Prima che il programma ha usato la funzione per la prima volta,l'entry contiene un indirizzo del run- time- linker(rtl). Se la funzione p chiamata dal programma il controllo è passato al rtl e l'indirizzo della funzione reale è stabilito e inserito nel GOT. Ogni chiamata a quella funzione passa il controllo direttamente a esso e il rtl non è più chiamato per questa funzione. Per una più completa overview su l'exploitation attraverso il GOT,riferirsi all'eccellente articolo di Lam3rZ[19].

Dalla sovrascrittura dell'entry del GOT per una funzione il programma userà dopo la vulnerabilità del format string è stata exploittata,possiamo ottenere il controllo e può saltare a ogni indirizzo eseguibile. Questo significa sfortunatamente che ogni protezione basata sullo stack,che compie un controllo sull'indirizzo di ritorno fallirà.

Il grande vantaggio che otteniamo con la sovrascrittura dell'entry GOT è la sua indipendenza dallo sviluppo delle variabili (per esempio lo stack) e l'allocazione dinamica della memoria (heap). L'indirizzo di un'entry GOT è solo stabilita per i binari,così se due sistemi utilizzano gli stessi binari,allora l'entry GOT è sempre allo stesso indirizzo.

Puoi vedere dove si trova l'entry GOT per una funzione utilizzando:

objdump - - dynamic- reloc binary

L'indirizzo della funzione reale (o la funzione rtl linking) è direttamente allo stesso indirizzo stampato.

Un altro importante fattore sul perché utilizzare delle GOT entries per ottenere il controllo invece degli indirizzi di ritorno è il codice del form(trovato in alcuni demoni finger 'sicuri'):

```
syslog (LOG_NOTICE, user);
exit (EXIT_FAILURE);
```

Non puoi ottenere il controllo qui con una sovrascittura dell'indirizzo di ritorno. Puoi provare a sovrascrivere il proprio indirizzo di ritorno del syslog,ma una via più facile è di sovrascrivere l'entry GOT della funzione 'exit',che passerà l'esecuzione all'indirizzo da te specificato tanto presto quanto 'exit' è chiamata.

Ma il vantaggio più utile della tecnica GOT è la sua facilità d'uso,devi solo eseguire objdump e hai l'indirizzo da sovrascrivere(retloc).

6.1.2 DTORS

Binari compilati con il compilatore GNU C includono una speciale sezione distruttore di tavola,chiamata 'DTORS'. I distruttori qui elencati sono chiamati giusto prima che sia emessa la reale 'exit' system call,dopo che tutte le normali operazioni di cleanup sono fatte. La sezione DTORS è del formato seguente:

```
DTORS: 0xffffffff 0x00000000 ...
```

Dove la prima entry è un contatore che tiene il numero dei puntatori alla funzione che seguono p meno uno (come in questo caso) se la lista è vuota.

Su tutte le implementazioni della sezione DTORS questo campo è ignorato. Allora,al relativo offset +4 ci sono gli indirizzi delle funzioni di cleanup,terminate da un indirizzo NULL. Puoi giusto sovrascrivere questo puntatore NULL con un puntatore alla tua shellcode e il tuo codice verrebbe eseguito ogni volta che il programma esce. Un più completo approfondimento può essere trovato a [17].

6.1.3 C library hooks

Alcuni mesi fa un Designer Solar introdusse una nuova tecnica per exploittare tramite un overflow basato su heap nella memoria allocate da malloc. Gli suggerì di sovrascrivere

un hook che è presente nella libreria GNU C ed altre librerie. Normalmente questo hook è utilizzato dal debugging della memoria e dai tools di profiling,per essere informato ogni volta che veniva allocata o liberata memoria usando l'interfaccia malloc. Ci sono pochi hook,ma i più comuni sono __malloc_hook, __realloc_hook e __free_hook. Normalmente sono settati a NULL,ma tanto presto quanto tu li sovrascrivi con un puntatore al tuo codice,il tuo codice verrà eseguito come sia che malloc,realloc o free siano chiamate.
Poiché gli hook sono normalmente utilizzati come debug hooks,sono chiamati prima che la funzione reale è chiamata.
Una discussione circa il malloc- overwrite è data dall'advisory [15].
6.1.4 strutture atexit.
Sempre pochi mesi fa,Kalou introdusse un modo per exploittare staticamente binari linkati sotto Linux,che prende vantaggio di un generico handler chiamato '__atexit',che riceve l'eseguito tanto presto quanto il programma chiama l'exit. Questo permette ad un programma di sistemare un numero di handler che sarà chiamato come esso escemper rilasciare risorse. Una discussione dettagliata può essere trovata su [16].
6.1.5 function pointers
Se l'applicazione vittima fa uso di puntatori a funzione,ci sono possibilità che tu le sovrascriva. Per fare uso di esse,devi sovrascriverle e poi azionarle. Alcuni demoni usano tavole di puntatori a funzione per elaborare un comando,per esempio QPOP. Anche puntatori a funzione sono spesso utilizzati per simulare un hanlder atexit - like,come in SSHd.
6.1.6 jmpbuf's
Le prime tecniche di sovrascrittura jmpbuf erano usate nelle exploitation di buffer situati negli heap. Con le format string jmpbuf si comporta come un puntatore a funzione,poiché possiamo scrivere ovunque nella memoria,senza la limitazione dalla posizione relativa di jmpbuf nel nostro buffer. Una profonda discussione può essere trovata in [18].
6.2 Ritorno dentro LibC
Puoi utilizzare la comune tecnica del ritorno nelle LibC,scoperte da –ancora una voltada Solar Designer[14]. Ma qualche volta potrebbe esserci uno shortcut,che risulta in una più facile exploitation:

```
FILE * f;
char foobuf[512];
snprintf (foobuf, sizeof (foobuf), user);
foobuf[sizeof (foobuf) - 1] = '\0';
f = fopen (foobuf, "r");
```

You can overwrite the GOT address of 'fopen' with the address of the 'system' function. Then use a format string such as:

```
"cd /tmp;cp /bin/sh .;chmod 4777 sh;exit;"
"addresses|stackpop|write"
```

Dove 'addresses','stackpop' e 'write' sono le comuni sequenze dell'exploitation delle format string. Sono usate per sovrascrivere l'entry GOT di 'fopen' con l'indirizzo di 'system'. Come 'fopen' è chiamato la stringa è passata alla funzione 'system'.
Alternativamente puoi usare il vecchio comune metodo,come descritto qui sotto:
6.3 Scrittura Multipla
Se puoi azionare più volte la vulnerabilità del format string dentro lo stesso processo (esempio in wu- ftpd),puoi sovrascrivere più che solamente l'indirizzo di ritorno. Per esempio puoi immagazzinare l'intero shellcode nell'heap per raggirare ogni protezione non eseguibile dello stack. Insieme con le altre tecniche spiegate qui tu puoi circoscrivere la seguente attrezzature di protezione (certamente non complete):
• StackGuard

• StackShield
• Openwall kernel patch (by Solar Designer)
• libsafe

A metà di ottobre 2000 un gruppo di persone pubblicò una serie di patch per il kernel di Linux conosciute come PaX[11],che effettivamente permettono di implementare pagine che sono leggibili e scrivibili ma non eseguibili. Poiché non è possibile fare questo originariamente sulle serie di CPU x86,la patch utilizza alcuni trucchi,scoperti da Plex progetto di emulatore di CPU. Su un sistema che utilizza questa patch è virtualmente impossibile di utilizzare shellcode arbitrariamente introdotto nel processo. Ma la maggior parte delle volte c'è già un codice utile dentro dello stesso spazio del processo. Possiamo eseguire questo codice per fare cose che faremmo normalmente nel nostro shellcode.

Usando le comuni tecniche del Return- into- LibC[14] puoi raggirare questa protezione. Il caso più semplice è ritornare nella funzione di libreria system() usando la format string come parametro.

Ottimizzando la stringa un pò,puoi ridurre gli offsets obbligatori da conoscere ad uno solo:l'indirizzo della funzione system(). Per chiamare un programma puoi usare questa sequenza alla fine della tua format string:

";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;id > /tmp/owned;exit;"

Ogni indirizzo che punta dentro il carattere ';' e passato alla funzione system() eseguira i comandi,poiché il carattere ';' attua comandi 'nop' alla shell.

6.4 Format string dentro l' Heap

Fino ad ora abbbiamo assunto che la format string giace sempre nello stack. Ma comunque,ci sono casi,,nei quali è immagazzinata nell'heap. Se c'è un altro buffer nello stack che possiamo influenzare,possiamo usare quello per fornire gli indirizzi nei quali scrivere,ma se non c'è abbastanza buffer ci rimangono poche alternative.

Se il buffer bersaglio giace nello stack,possiamo prima scriverci,e poi usare gli indirizzi da lì,per scrivere usando i parametri '%n':

```
void func (char *user_at_heap)
{
char outbuf[512];
snprintf (outbut, sizeof (outbuf), user_at_heap);
28 6 SPECIAL CASES
outbuf[sizeof (outbuf) - 1] = '\0';
return;
}
```

Qui usiamo una format string che contiene gli indirizzi nei quali vogliamo scrivere,come al solito. Ma la cosa speciale a suo riguardo,è che non accediamo a quei indirizzi dalla stessa format string,ma dal buffer bersaglio. Per fare questo dobbiamo prima immagazzinare gli indirizzi nello stack,semplicemente scrivendoli. Dunque la sequenza di scrittura deve essere dietro gli indirizzi dentro la format string.

Se entrambi i buffer non giacciono nello stack,abbiamo un problema:

```
void func (char *user_at_heap)
{
char * outbuf = calloc (1, 512);
snprintf (outbut, 512, user_at_heap);
outbuf[511] = '\0';
return;
}
```

Ora dipende se abbiamo qualche modo di fornire dati nello stack. Per esempio,alcuni exploits utilizzarono per wu- ftpd il campo della password per immagazzinarci data (shellcode,non indirizzi tuttavia –quegli exploits mon possono exploittare un account

non anonimo).
Ogni vulnerabilità ed explolit è differente,e dovrebbe investire ore per studiare la vulnerabilità prima di constatare che non è exploitabile,e anche allore ci sono casi che tu svagli,come non solo la storia delle vulnerabilità del format string ha dimostrato.

6.5 Considerazioni Speciali

Dietro la stessa exploitatation ci sono alucune cose da considerare. Se lo shellcode è contenuto dentro la format string potrebbe non contenere '\x25'(%) o NULL bytes. Ma poiché nessun importante opcode è ne 0x25 ne 0x00 non incontrerai problemi quando costruirai lo shellcode. Lo stesso è vero se gli indirizzi sono anche immagazzinati nella format string. Se un indirizzo che vuoi che scriva per contenere un NULL byte nel meno significante byte,puoi sostituirlo con uno short - write in un indirizzo occasionale appena sotto l'indirizzo su cui vuoi immagazzinare il byte. Tuttavia questo non è possibile su tutte le architetture. Puoi anche usare due format string separate. La prima crea l'indirizzo in cui vuoi scrivere nella memoria dietro la stringa intera. La seconda utilizza questo indirizzo per scriverci. Questo potrebbe diventare complicato piuttosto presto,ma permette exploitation fidate e qualche volta ne vale la fatica.

7 Tools

Una volta che l'exploit è finito,o perfino nel momento dello sviluppo dell'exploit è di aiuto usare tools per ricavare gli offsets necessari. Alcuni tools possono anche aiutare nell'identificazione di vulnerabilità,per esempio la vulnerabilità del format string in un codice chiuso di un software(closed source software).Ho listato 4 tools qui,che mi sono stati di grande aiuto e potrebbe esserlo anche per te.

7.1 ltrace, strace

ltrace [8] e strace [9] lavorano in un modo simile:essi agganciano chiamate alla libreria e al sistema,loggando i loro parametri e i loro valori,come il programma li chiama. Questo ti permette di osservare come il programma interagisce col sistema,considerando lo stesso programma come una black box.
Tutte le funzioni già fatte sono chiamate alle librerie e i loro parametri,molto importanti per i loro indirizzi,possono essere osservato usando ltrace. In questo modo puoi velocemente determinare l'indirizzo della format string in ogni processo,you can ptrace (non sono riuscito a tradurloJ). Il programma strace è usato per ricevere gli indirizzi dei buffer dove dentro sono letti i dati,per esempio se read è chiamato per leggere dati che che sono usati dopo come format string.
Imparare ad utilizzare questi due tools può ripagare salvando molto tempo,che utilizzeresti provando ad attaccare GDB ad un programma outdated con simboli che perdi e ottimizzazione del compilatore,solo per trovare due offset.

7.2 GDB, objdump

GDB [7], il classico 'GNU Debugger' è un debugger testuale,che è capace sia di fare il debugging del source level e del codice macchina. Sebbene non appare molto confortevole,una volta che hai imparato ad utilizzarlo,è una efficace interfaccia per gli inernals dei programmi,è utile per ogni cosa dal debugging del tuo exploit al guardare come il processo diventa exploittato.
Objdump,un programma per il pacchetto binutils GNU,è capace di ricevere ogni informazione da un binario eseguibile o da un object file,per esempio il layout della memoria,le sezioni o il disassemblaggio delle funzioni principali. Lo usiamo principalmente per ricevere gli indirizzi delle entry GOT dal binario. Ma può servirti per molte altre code utili.

8 Referenze

[1] TESO Security Group,
http: / /www.team- teso.net /
[2] Chaos Computer Club: 17th Chaos Communication Congress,
http: / /www.ccc.de/congress/

[3] portal,
“Format String Exploitation Demystified”, preliminary version 21,
not yet published, http: / /www.security.is/
[4] Pascal Bouchareine,
“format string vulnerability”,
http: / /www.hert.org/papers/ format.html
[5] Plasmoid / THC,
Stack overflows,
http: / /www.thehackerschoice.com/papers/OVERFLOW.TXT
[6] Halvar Flake,
“Auditing binaries for security vulnerabilities”,
http: / /www.blackhat.com/presentations/bh- europe-
00/HalvarFlake/HalvarFlake.ppt
[7] GDB, The GNU Debugger,
http: / /www.gnu.org/software/gdb/gdb.html
[8] ltrace, no official maintainer,
http: / /www.debian.org/Packages/stable/utils/ l t race.html
[9] strace,
http: / /www.wi.leidenuniv.nl /%7ewichert /strace/
[10] GNU binutils,
http: / /www.gnu.org/gnulist /production/binutils.html
[11] PaX group,
“Implementing non executeable rw pages on the x86”,
http: / /pageexec.virtualave.net /
[12] Tool Interface Standard,
Executeable and Linking Format Specifications v1.2,
http: / /segfault.net /%7escut /cpu/generic/TIS- ELF v1.2.pdf
[13] Silvio,
“ELF executeable reconstruction from a core image”,
http: / /www.big.net.au/%7esilvio/core- reconstruction.txt
REFERENCES 31
[14] Solar Designer,
post to Bugtraq mailing list demonstrating return into libc,
Bugtraq Archives 1997 August 10
[15] Solar Designer,
JPEG COM Marker Processing Vulnerability in Netscape Browsers,
advisory demonstrating malloc management information overwrite,
http: / /www.openwall.com/advisories/OW- 002- netscape- jpeg.txt
[16] Pascal Bouchareine,
“__atexit in memory bugs: proof of concept”
[17] Juan M. Bello Rivas,
“Overwriting the .dtors section”
[18] Matt Conover aka Shok,
“w00w00 on Heap Overflows”,
http: / /www.w00w00.org/ f iles/articles/heaptut.txt
[19] Bulba and Kil3r, Lam3rZ,
Bypassing StackGuard and StackShield,
Phrack issue 56, article #5, http: / /phrack.infonexus.com/
[20] Kil3r, Lam3rZ,
33_su.c, exploit for su/msgfmt for Immunix Linux
[21] LSD crew,
IRIX telnet daemon exploit irx_telnetd.c and explanations,
http: / /www.lsd- pl.net /,

http: / /www.securityfocus.com/ templates/archive.pike?list=1&mid=75864
[22] TESO wu- ftpd 2.6.0 exploit: 7350wu,
http: / /www.team- teso.net /releases.php

9 Conclusioni

Spero che vi sia gustata questa traduzione,anche perchè mi è costata un sacco fatica.
Ciao a tutti!
Styx^