

# Sicherheitsrelevante Programmierfehler (Teil 3)

[Thomas Biege](#)

## Inhaltsverzeichnis

- [Temporäre Dateien](#)
- [Format Bugs](#)

## Temporäre Dateien

Neben Speicherüberläufen sind Fehler bei der Handhabung temporärer Dateien der häufigste Grund für Sicherheitsprobleme.

In der Regel werden temporäre Dateien in **öffentlich beschreibbaren Verzeichnissen** angelegt (Unix: /tmp, /var/tmp). Um das Löschen von Dateien in öffentlichen Verzeichnissen zu verhindern, wird beispielsweise in Unix ein **Filesystem-Flag** gesetzt (chmod o+t /tmp), somit kann ein Benutzer nur seine Dateien aus dem Verzeichnis entfernen. Unterverzeichnisse werden nicht von dem Flag geschützt, d.h. wenn das Unterverzeichnis für jederman beschreibbar ist, dann kann in diesem Verzeichnis alles verändert werden. Das **t-Flag** muß also für jedes Verzeichnis explizit gesetzt werden. Ein Beispiel für diesen Fehler ist das Verzeichnis /tmp/soffice.tmp, das von StarOffice erstellt wird.

Neben dem unerlaubten Löschen von Dateien besteht noch die Gefahr des Lesens von geheimen Daten (z.B. bei älteren *Acrobat Reader* und *WordPerfect* Versionen) und **Race Conditions** oder **Angriffe über Filelinks**. Folgende **nicht-atomare** und **unsichere Codesegmente** zur Erstellung von temporären Dateien werden oft benutzt:

Fehler:

```
[...]
FILE *tmp_datei;
[...]

tmp_datei = tmpfile();
[...]
```

oder:

```
[...]
char *tmp_name;
int tmpfd;

tmp_name = tmpnam(NULL);

if( (tmpfd = open(tmp_name, O_RDWR | O_CREAT)) < 0)
    [EXIT]
unlink(tmp_name);
/*
** entferne den Dateinamen, damit bei Beendigung des
** Programmcodes oder beim Schließen des Filedeskriptors
** die Datei nicht auf dem Filesystem zurückbleibt.
**
*/
```

```
[...]
```

Beide Methoden zur Dateierzeugung sind unsicher, da sie nicht atomar sind und symbolischen Links folgen. Im zweiten Beispiel ist nicht die Funktion *tmpnam(3)* für das Sicherheitsrisiko verantwortlich, sondern der darauffolgende *open(2)* Befehl. Funktionen wie *tmpnam(3)*, *tempnam(3)* oder *mktemp(3)* garantieren nur, daß der Dateiname zum Zeitpunkt ihres Aufrufs nicht vorhanden ist. Aber zwischen der Erzeugung des Namens und dem Öffnen kann ein Angreifer beispielsweise einen Link mit genau diesem Namen erzeugen und *open(2)* folgt dem Link (sog. **Race Condition**).

Korrektur:

```
[...]  
int tmpfd;  
[...]  
  
if( (tmpfd = mkstemp("/tmp/MyTempFile.XXXXXX")) < 0)  
    [Exit]  
  
fchmod(tmpfd, 0600);  
[...]
```

*Mkstemp(3)* erzeugt einen eindeutigen Namen und öffnet ihn auf sichere Art und Weise. Leider hat diese Lösung den Nachteil, daß *mkstemp(3)* nicht im *POSIX* Standard enthalten ist (*BSD4.3*) und alte Versionen die Zugriffsrechte auf 0666 setzen, so daß jeder in die Datei schreiben und daraus lesen kann. Wenn man sicheren und portablen Code erzeugen möchte, dann muß man selbst *open(2)* mit den richtigen Parametern aufrufen.

```
[...]  
int tmp_fd;  
FILE *tmp_stream;  
char tmp_name;  
[...]  
  
if((tmp_name = tmpname(NULL)) == NULL)  
    [Exit]  
  
if((tmp_fd = open(tmp_name, O_RDWR|O_CREAT|O_EXCL, 0600)) < 0)  
{  
    fprintf(stderr, "Possible Link Attack detected!\n");  
    exit(-1);  
}  
  
/*  
  
** Wir wollen die Stream-I/O Funktionen aus stdio.h  
** nutzen.  
*/  
if( (tmp_stream = fdopen(tmp_fd, "rw")) < 0)  
    [Exit]  
[...]
```

## Format Bugs

Ein neuer Bugtyp, der ebenfalls zu Sicherheitsproblemen führt, ist der sog. Format(-ing) Bug. Dieser Bug wurde erst vor kurzem entdeckt und schlummert schon seit vielen Jahren unbemerkt in vielen Softwarepaketen.

Betroffen sind alle Programme, die Daten aus nicht-vertrauensvollen Quellen an Funktionen mit

variabler Parameterlänge als *Format-String* weitergeben. (Also alle *printf(3)*-ähnlichen Funktionen).

Fehler:

```
[...]
snprintf(buf, sizeof(buf), UntrustedUserDataBuffer);
[...]
```

Korrektur:

```
[...]
snprintf(buf, sizeof(buf), "%s", UntrustedUserDataBuffer);
[...]
```

Hier kann natürlich auch *strncat(3)* o.ä. benutzt werden.

Um den Bug besser verstehen zu können, muß man wissen, wie die variable Anzahl von Parametern in C gehandhabt wird.

Die nicht-fixe Parameteranzahl wird bei der Funktionsdeklaration einfach mit drei Punkten "..." definiert.

Beispiel:

```
int my_sprintf(char *buffer, char *format_string, ...);
```

Die unbekannte Anzahl der übergebenen Argumente kann die Funktion mit den Makros *va\_start(3)*, *va\_arg(3)* und *va\_end(3)*, die in *stdarg.h* definiert sind, bearbeiten.

Beispiel:

```
int my_sprintf(char *buffer, char *format_string, ...)
{
    va_list arg_ptr; /* Argumentzeiger */
    short ShortValue;
    [...]

    va_start(arg_ptr, format_string);
    /*
    ** setze arg_ptr auf das erste optionale
    ** Argument.
    */
    [...]

    while(format_string != NULL)
    {
        [...]

        ShortValue = va_arg(arg_ptr, short);
        /* va_arg() liefert den short Wert */
        [...]

        ++format_string;
        /* naechstes Zeichen im Format-String */

    }
    va_end(arg_ptr);
    [...]
}
```

Um an den nächsten optionalen Parameter zu gelangen, geht *va\_arg(3)* einfach den Stack rückwärts hoch (Adressen werden dekrementiert). Wenn der Angreifer die Möglichkeit hat, den *Format-String*

selbst zu definieren, kann er *Format-Tags* innerhalb des *Format-Strings* angeben, ohne daß dafür *Format-Variablen* vorhanden sind.

Beispiel:

```
[...]
#define MAX_BUF 1024;
[...]

char Buffer[MAX_BUF];
char UntrustedUserDataBuffer[MAX_BUF];
[...]

getDataFromNetwork(UntrustedUserDataBuffer, MAX_BUF);
[...]

my_sprintf(Buffer, UntrustedUserDataBuffer);
/*
** UntrustedUserDataBuffer enthält beispielsweise
** "Gute Nacht! %s %p %p %s %p"
** Für die Format-Tags existiert keine Format-Variable!
*/
[...]
```

Auch wenn keine *Format-Variablen* vorhanden sind geht *va\_arg(3)* einfach den Stack hinauf, um die Werte für die Platzhalter zu bekommen. Es werden natürlich auch die *Stack-Frames* von anderen Funktionen durchsucht. Aufgrund dieser Eigenschaft ist es dem Angreifer nicht nur möglich, das Programm zum Absturz zu bringen, er kann auch geheime/wertvolle Informationen lesen, den Wert lokaler Variablen ändern oder sogar eigenen Code ausführen, indem er Funktionszeiger, Sprungadressen oder gesicherte IP-Werte manipuliert. Um Veränderungen innerhalb des Prozeßspeichers durchführen zu können, ist die *Format-Variable* "%n" nötig, mit deren Hilfe die (nicht-reale) Position im *Format-String* an eine bestimmte Adresse, die durch eine *Format-Variable* angegeben wird, geschrieben wird..

Folgendermaßen können *Format Bugs* vermieden werden:

```
my_sprintf(Buffer, "%s", UntrustedUserDataBuffer);
```

Somit werden die *Format-Tags* nicht nochmal geparkt.