

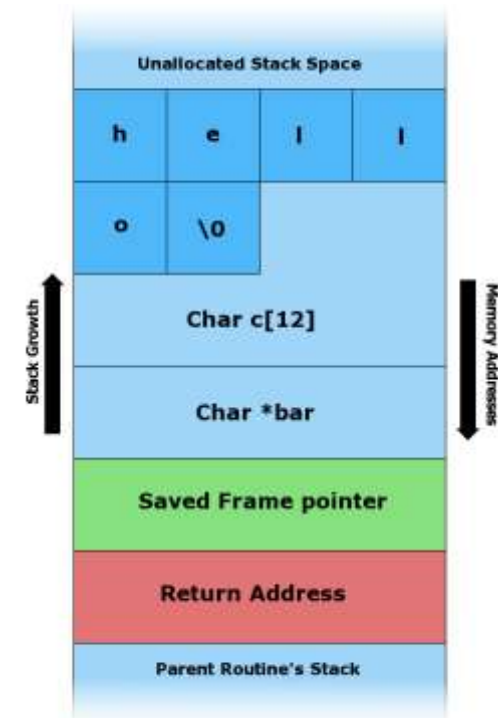
# Grundlagen der IT-Sicherheit

## VL 7: Softwaresicherheit 2



# Unsere heutigen Themen...

- Buffer Overflow
  - Stack Overflow
  - Heap Overflow
  - Integer Overflow
- Datenmanipulation
  - Read Overflow
  - Heart Bleed
  - Stale Memory
  - Format-String Verwundbarkeiten
- Schutzmassnahmen
  - Memory safety
  - Canaries

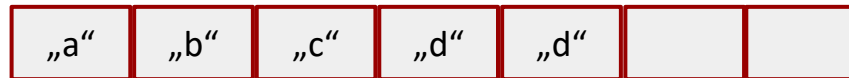


# BUFFER OVERFLOW

Überblick, Ursachen, Stack Overflow und Heap Overflow

**Buffer:** Ein Array mit fester Länge und gleichem Datentyp

Character Array mit Länge 5



Stack: Lokale Variablen

```
char buf[42];
```

Heap: Pointer zu einem Block

```
char *buf = malloc(42);
```

**Buffer Overflow:** Über das Ende eines Buffers hinausschreiben

Character Array mit Länge 5



Folgende Daten und Metadaten werden überschrieben.

Geschriebene Daten werden durch Angreifende kontrolliert.

**Buffer Overflow:** Über das Ende eines Buffers hinausschreiben

**Exploitability** (Ausnutzbarkeit für einen Angriff):

- Abhängig von verschiedenen Faktoren, darunter
  - Lage des Speichers (Stack oder Heap)
  - CPU Architektur und Speicherlayout

# Buffer Overflow: Root Cause

## Unzureichende Speicherzuweisung

- Viele EntwicklerInnen sind sich des Overflow-Problems nicht bewusst
- Viele unbeschränkte String-Funktionen: `gets, strcpy, sprintf, ...`
  - Beispiel Heartbleed!

## Fehler bei der Berechnung von Größe und Speicherort

- Integer und Arithmetische Overflows
- Typumwandlungen („casting errors“)
- Gefährliche String-Funktionen:

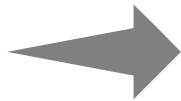
```
memcpy(buf, str, 8);
```

```
strcpy(buf, "woah!");
```

```
sprintf(buf, "%.9f", x);
```

```
0x7FFFFFFF + 1
```

# Buffer Overflow: Verschiedene Varianten

- 
- Stack Overflow
    - Heap Overflow
    - Integer Overflow

## Verwundbare C function

```
void check(char *password) {  
    int a = 42;  
    char buf[10];  
    strcpy(buf, password);  
}
```

## Let's overflow it

```
void main() {  
    check("abcdefghijklm");  
    return;  
}
```

check(char \*password) ist (wahrscheinlich) verwundbar, da es die strcpy Funktion verwendet.

### Description

The **strcpy()** function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)

Eine Suche nach strcpy auf GitHub gibt > 1 M Ergebnisse!



Exploit eines Stack Overflows:

- Aktuelles und vorheriges Stack-Frame überschreiben
- Lokale Variablen und/oder Rücksprungadressen überschreiben

Manipulation lokaler Variablen

- Kontrollfluss umlenken durch Zuweisung von Variablen

Manipulation der Rücksprungadresse

- Ausführbaren Code einschleusen (aka *Shellcode*)
- Rücksprungadresse (aka RET) umlenken zum eingeschleusten Code

# Stack Overflow: Angriff auf lokale Variablen

## Verwundbare auth function

```

void auth(char *arg) {
    int auth = 0;
    char pass[10];
    strcpy(pass, arg);

    if(!strcmp(pass, "<5#)z02="))
        auth = 1;

    if(auth != 0)
        printf("logged in");
    else
        printf("login error");
}

```

## Exploit

```
auth("aaaaaaaaaaaaa");
```

## Stack frame nach strcpy

```

0x5312 arg
0x1004 RET
0x7ff8 BP
0x4100 auth
0x4141
0x4141
0x4141
0x4141
0x4141 pass

```



pass over-  
flows auth



auth != 0

# Stack Overflow: Manipulation der Rücksprungadresse

## Verwundbare auth function

```
void auth(char *arg) {
    int auth = 0;
    char pass[128];
    strcpy(pass, arg);

    if(!strcmp(pass, "<5#)z02="))
        return;

    ...
}
```

## Exploit

```
// NOPs + Shellcode + new RET
auth("\x90\x09...\x7f\x32");
```

## Manipulation des Kontrollflusses

## Stack frame nach strcpy

```
0x5312 arg
0x7f32 RET
0xffff BP
0xffff
...
...
0x5e89
0x9090
0x9090 pass
```

Manipuliere  
RET



RET = 0x7f32

## Shellcode: Code zum Ausnutzen einer Sicherheitslücke

- Ursprünglich verwendet um eine Befehlszeile (shell) zu starten
- Unterschiedliche Arten: Blindshell, Staged Shellcode, usw.

```
/* ASSEMBLY
31 db          xor     %ebx,%ebx
b0 17          mov     $0x17,%al
cd 80          int     $0x80
31 c0          xor     %eax,%eax
50            push    %eax
68 61 64 6f 77 push    $0x776f6461
68 63 2f 73 68 push    $0x68732f63
68 2f 2f 65 74 push    $0x74652f2f
89 e3          mov     %esp,%ebx
66 b9 b6 01     mov     $0x1b6,%cx
b0 0f          mov     $0xf,%al
cd 80          int     $0x80
40            inc     %eax
cd 80          int     $0x80
*/

int main(){
    char shell[] = "\x31\xdb\x00\x17\xcd\x80\x31\x00\x50"
"\x68\x61\x64\x6f\x77\x68\x63\x2f\x73\x68"
"\x68\x2f\x2f\x65\x74\x89\xe3\x66\xb9\xb6\x01"
"\xb0\x0f\xcd\x80\x40\xcd\x80";

    printf("[*] Taille du ShellCode = %d\n", strlen(shell));
    (*(void (*)()) shell)();

    return 0;
}
```

Beispiel-Shellcode:

Manipuliert Zugriffskontrolle auf  
etc/shadow

<http://shell-storm.org/shellcode/files/shellcode-608.php>


# Beispiel Stack Overflow: SQL Slammer (2003)

- Exploited einen Stack Overflow in Microsofts SQL Server
- Geschädigte wurden meist in unter 10 Minuten infiziert



[https://media.scmagazine.com/images/2017/02/03/sqlslammer\\_1150643.jpg](https://media.scmagazine.com/images/2017/02/03/sqlslammer_1150643.jpg)

# Buffer Overflow: Verschiedene Varianten

- 
- Stack Overflow
  - Heap Overflow
  - Integer Overflow

## Verwundbare C function

```
void check(char *password) {  
    char *buf;  
    buf = malloc(4)  
    strcpy(buf, password);  
}
```

## Let's overflow it

```
check("abcdefghijklmop");
```

Before *strcpy*



After *strcpy*



## Exploit eines Heap Overflows

- *Nächsten* Heap Block und *seine Metadaten* überschreiben
- Listenpointer manipulieren
- Exploit ist stark abhängig von der Implementierung!

## Unlink-Angriff

- Manipulation der Listenpointer um an eine frei wählbare Adresse zu schreiben
- Wird ausgelöst beim Zusammenführen (merging): Aufruf von *free()*



# Heap Overflow: Unlink-Angriff

Snippet from free()

```
b->next->next->prev = b->next->prev
```

target address ← value




Umleitung des Kontrollflusses

- Manipulation der Rücksprungadresse
- Manipulation des Ausnahme-Handling
- Manipulation von Wartungsfunktionen

Herausforderung: Genaue Speicheradresse muss bekannt sein

# Buffer Overflow: Verschiedene Varianten

- Stack Overflow
- Heap Overflow
-  Integer Overflow

Idee: Eine zu große Nummer in einer Integer-Variable speichern

**Annahme: 16 Bit Integer**

Maximale Größe:  $2^{16} = 65.536$

Was passiert, wenn wir 65.537 in eine 16 Bit Integer-Variable speichern?

Undefiniertes Verhalten in C / C++!



Annahme: 32 Bit Integer

```
void vulnerable()
{
    char *response;
    int nresp = packet_get_int();
    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
}
```

Was passiert, wenn wir  $nresp=1.073.741.824$  setzen?

- Annahme:  $\text{sizeof}(\text{char}^*) = 4$
- Malloc'd:  $4 \cdot 1.073.741.824 = 4,294,967,296 = 2^{32}$

Die for()-Schleife erzeugt einen Overflow!

# Umgang mit Integer Overflows

Language	Unsigned integer	Signed integer
Ada	modulo the type's modulus	<b>raise</b> <code>Constraint_Error</code>
C/C++	modulo power of two	undefined behavior
C#	modulo power of 2 in unchecked context; <code>System.OverflowException</code> is raised in checked context <sup>[1]</sup>	
Java	N/A	modulo power of two
JavaScript	all numbers are double-precision floating-point	
Python 2	N/A	convert to long type (bigint)
Seed7	N/A	<b>raise</b> <code>OVERFLOW_ERROR</code> <sup>[2]</sup>
Scheme	N/A	convert to bigNum
Smalltalk	N/A	convert to LargeInteger
Swift	Causes error unless using special overflow operators. <sup>[3]</sup>	

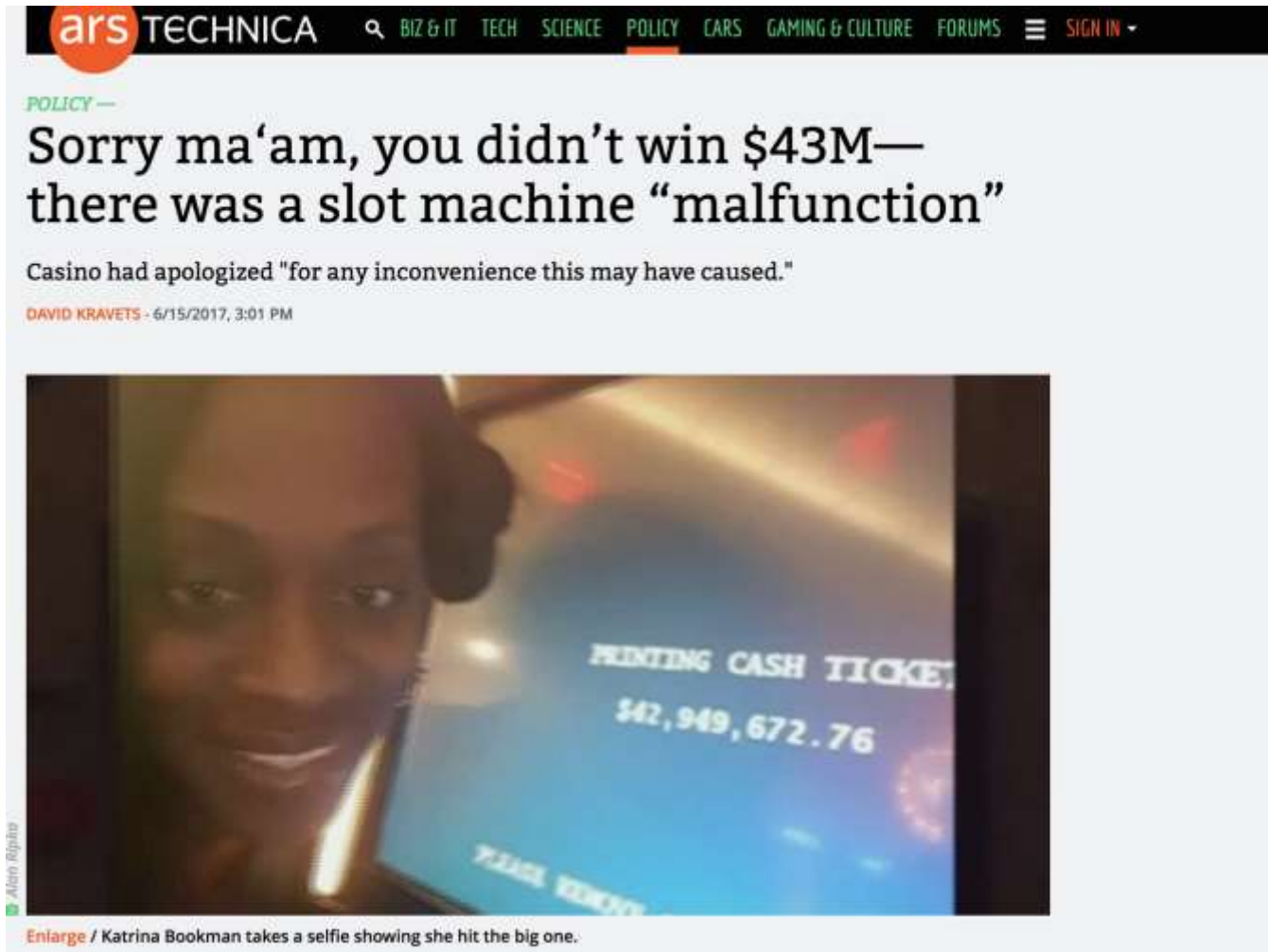
Unterschiedliche Strategien, mit Integer Overflows umzugehen

Meist wird Performance geopfert um für mehr Sicherheit zu sorgen.

Python is safe! :-)

[https://en.wikipedia.org/wiki/Integer\\_overflow](https://en.wikipedia.org/wiki/Integer_overflow)

# Beispiel Integer Overflow: Gewinne bei Glücksspielautomaten



<https://arstechnica.com/tech-policy/2017/06/sorry-maam-you-didnt-win-43m-there-was-a-slot-machine-malfunction/>



# DATENMANIPULATION

Überblick, Read Overflow, Stale Memory, Format String  
Verwundbarkeiten

Bisher haben die Angriffe Code betroffen ...

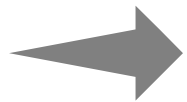
(Rücksprungadressen und Funktionspointer)

... doch Angriffe können sich auch auf **Daten** auswirken um ...

- **Einen geheimen Schlüssel zu modifizieren**, sodass man den Schlüssel kennt und zukünftige abgefangene Nachrichten entschlüsselt werden können
- **Zustandsvariablen zu modifizieren**, um Autorisierungsüberprüfungen zu umgehen (früheres Beispiel mit „authenticated flag“)
- **Interpretierbare Zeichenketten zu modifizieren**, die als Teil von Befehlen verwendet werden
  - Beispiel: SQL-Injection erleichtern, wird in späterer Vorlesung besprochen



# Datenmanipulation: Verschiedene Varianten



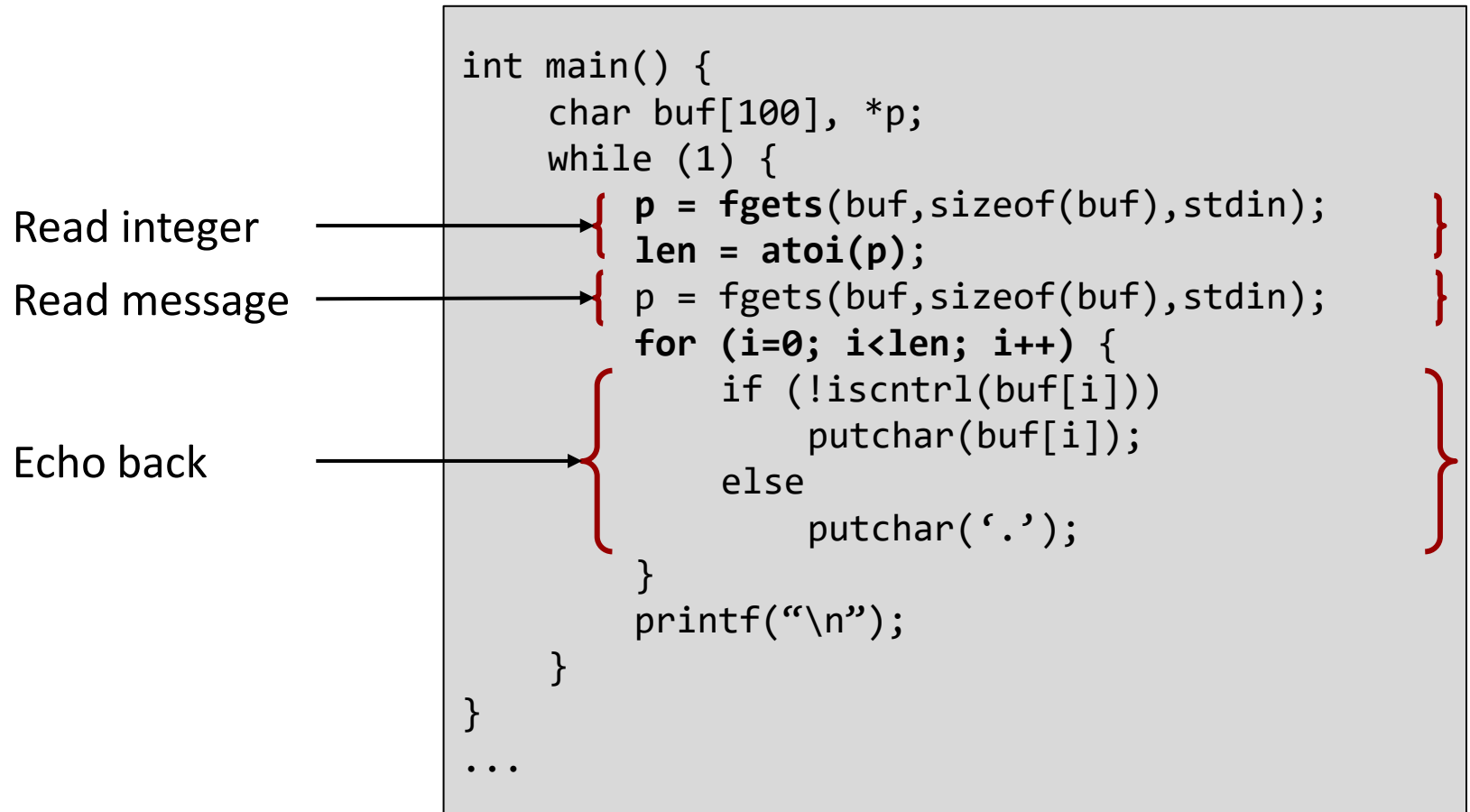
## Read Overflow

- Beispiel: Heart Bleed
- Stale Memory
- Format-String Verwundbarkeiten

Anstatt das **Schreiben über das Ende eines Buffers hinaus** zu erlauben (Buffer Overflow), könnte ein Fehler das **Lesen über das Ende hinaus** erlauben.

⇒ Man liest mehr Daten als man eigentlich dürfte.

⇒ Könnte geheime Informationen (Passwörter, geheime Dateien, uws.) preisgeben



# Read Overflow: Testlauf

```
% ./echo-server
```

```
24
```

```
every good boy does fine
```

```
ECHO: |every good boy does fine|
```

```
10
```

```
hello there
```

```
ECHO: |hello ther|
```

```
25
```

```
hello
```

```
ECHO: |hello..here..y does fine.|
```

OK: Input len < buffer size

BAD: len > size

# Datenmanipulation: Verschiedene Varianten

- Read Overflow
- ➔ Beispiel: Heart Bleed
- Stale Memory
- Format-String Verwundbarkeiten



- Das Beispiel kennen wir schon 😊
- Ein berühmtes Beispiel für einen Read Overflow
- Betroffene OpenSSL-Versionen: 1.0.1 through 1.0.1f
- Kennung der Schwachstelle: CVE-2014-0160
- Robin Seggelmann, ein in Deutschland ansässiger Programmierer, reichte den Code in einem Update ein, das am Silvesterabend 2011 um 23:59 Uhr submitted wurde.
- Meldung der Schwachstelle (Silent Report) am 1. April 2014
- Einfacher Overflow aufgrund fehlender Prüfung der Schranken
- Betroffene Dateien: *t1\_lib.c* und *d1\_both.c*

# Verwundbarkeit: Read Overflow

```
diff --git a/ssl/dl_both.c b/ssl/dl_both.c
index f0c5962..d8bcd58 100644 (file)
--- a/ssl/dl_both.c
+++ b/ssl/dl_both.c
@@ -1330,26 +1330,36 @@ dtls1_process_heartbeat(SSL *s)
     unsigned int payload;
     unsigned int padding = 16; /* Use minimum padding */

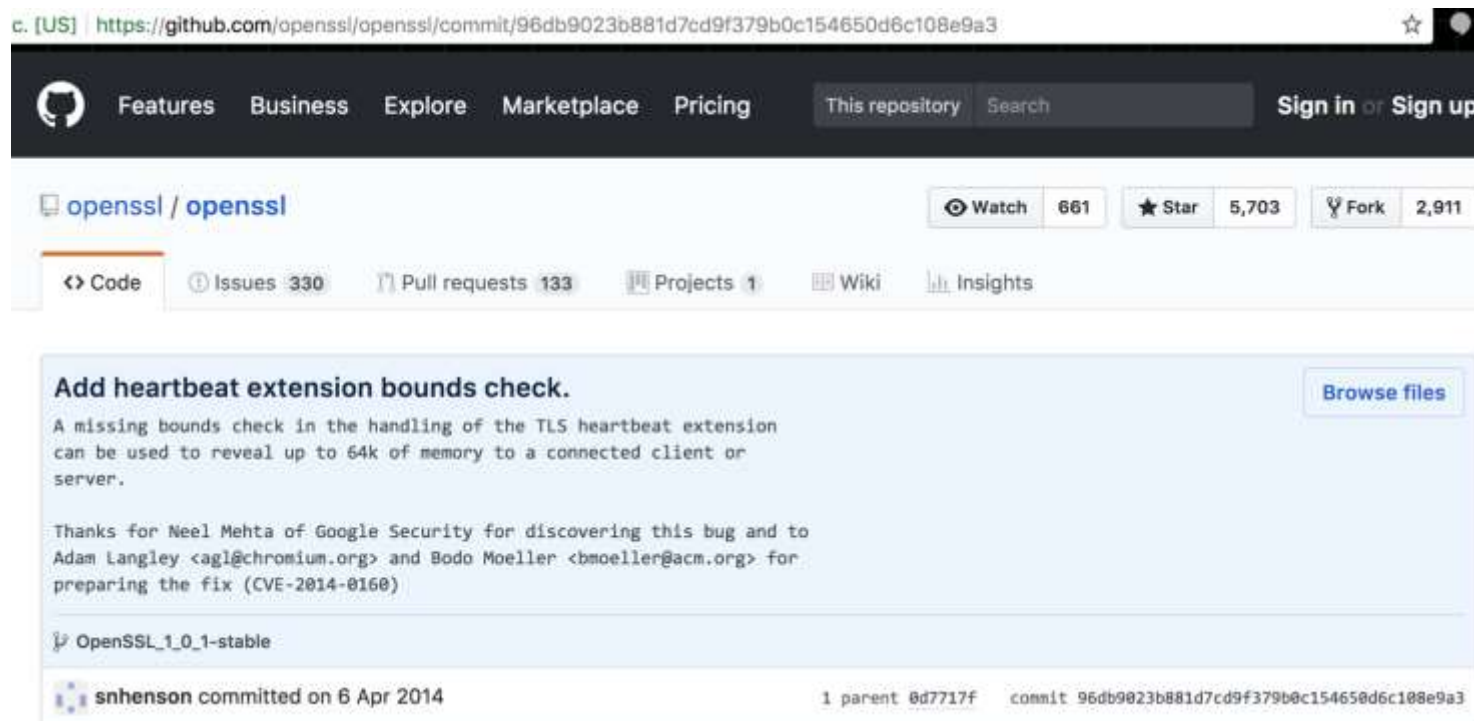
-    /* Read type and payload length first */
-    hbtype = *p++;
-    n2s(p, payload);
-    pl = p;
-
     if (s->msg_callback)
         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);
```

Fehlende Prüfung der Schranken

<https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=731f431497f463f3a2a97236fe0187b11c44aeadd>

Ignoriere die Heartbleed-Anfragen, die mehr Daten anfordern als ihr Payload benötigt:

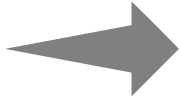
```
if (1 + 2 + payload + 16 > s->s3->rrec.length) return 0;
```





# Datenmanipulation: Verschiedene Varianten

- Read Overflow
- Beispiel: Heart Bleed
- Stale Memory
- Format-String Verwundbarkeiten



- „stale“ = „abgestanden, verbraucht, abgegriffen, alt“
- Ein **Dangling-Pointer-Fehler** tritt auf, wenn ein Zeiger freigegeben wird, das Programm ihn aber weiterhin verwendet.
- Angreifende können dafür sorgen, dass der **freigegebene Speicher neu zugewiesen** wird und unter deren Kontrolle steht.
  - Wenn der Dangling Pointer dereferenziert wird, greift er auf Daten unter der Kontrolle der Angreifenden zu.

```
struct foo { int (*cmp)(char*,char*); };  
struct foo *p = malloc(...);  
free(p);  
.  
.  
.  
q = malloc(...) //reuses memory  
*q = 0xdeadbeef; //attacker control  
.  
.  
.  
p->cmp("hello","hello"); //dangling ptr
```

Wenn Dangling-Pointer dereferenziert wird, wird er auf Daten zugreifen, die unter der Kontrolle der Angreifenden stehen

## IE's Role in the Google-China War



By Richard Adhikari  
TechNewsWorld  
01/15/10 12:25 PM PT

AA Text Size  
Print Version  
E-Mail Article

**The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the**

**exploit. The attack appears to consist of several files, each a different piece of malware.**

Computer security companies are scurrying to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

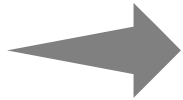
The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

### Pointing to the Flaw

The vulnerability in IE is an invalid pointer reference, Microsoft said in [security advisory 979352](#), which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

# Datenmanipulation: Verschiedene Varianten

- Read Overflow
- Beispiel: Heart Bleed
- Stale Memory



Format-String Verwundbarkeiten

- Fokus: *printf*-Funktionsfamilie in C
- Formatspezifikationen, Liste von Argumenten
  - Der „Specifier“ gibt den Typ des Arguments an (%s, %d, usw.)
  - Position der Zeichenkette gibt das zu druckende Argument an

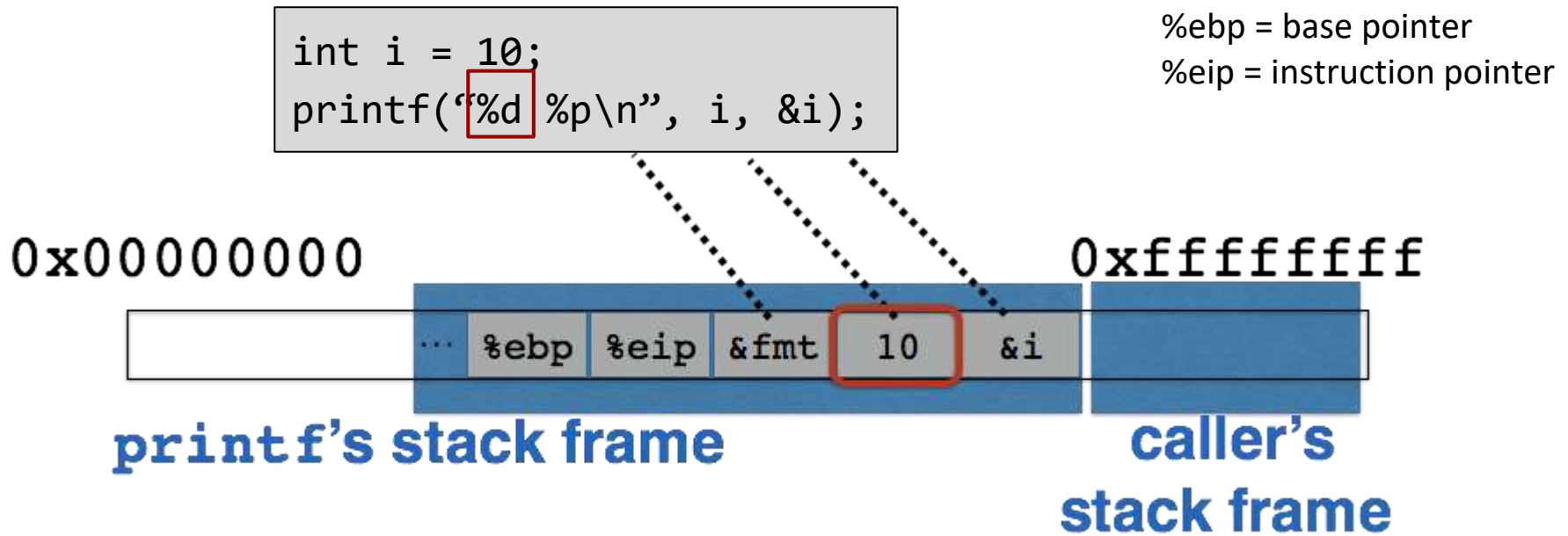
```
void print_record(int age, char *name)
{
    printf("Name: %s\tAge: %d\n", name, age);
}
```

# Wie kann das gefährlich sein?

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf); ← Angreifende kontrollieren Format-String
}
```

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s", buf);
}
```

# Implementierung von *printf*



- *printf* nimmt eine variable Anzahl von Argumenten
- Weiß nicht, wo das Stack-Frame „endet“
- Liest weiter vom Stack, bis der „out of format“ Spezifizierer kommt.

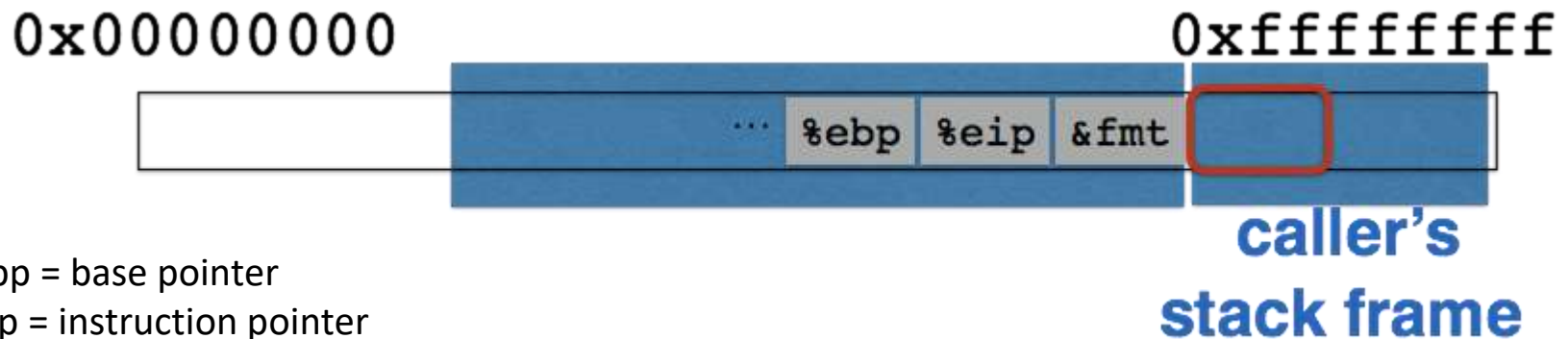


# Implementierung von *printf*

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

← **Attacker controls format string**

buf = “%d %x”



# Warum ist das ein Buffer Overflow?

Denkweise:

- Der Stack selbst kann als eine Art Buffer betrachtet werden
- Die Größe des Buffers wird durch die Anzahl und Größe der an sie übergebenen Argumente bestimmt

Die Bereitstellung eines falschen Format-Strings veranlasst also das Programm dazu, diesen „Buffer“ zu überlaufen.

# SCHUTZMAßNAHMEN

Was haben alle bisher betrachteten Angriffe gemeinsam?

- Angreifende können **bestimmte Daten kontrollieren**, die das Programm verwendet
- Die Verwendung dieser Daten ermöglicht **unerwünschten Zugriff auf bestimmte Speicherbereiche** des Programms
  - Über die Länge eines Buffers hinaus
  - Auf bestimmte Positionen auf dem Stack / Heap

Idee: Schutzmaßnahmen einbauen, um solche Angriffe zu verhindern



## Memory und Type Safety

„Speichersicherheit“ und „Typsicherheit“

Eigenschaften, die, wenn sie erfüllt sind, sicherstellen, dass eine Anwendung immun gegen Speicherangriffe ist!

Eine speichersichere Programmausführung:

- Erzeugt nur Zeiger mit **standardisierten Hilfsmitteln**  
`p = malloc(...)` oder `p = &x`, oder `p = &buf[5]`, usw.
- Verwendet einen Zeiger nur, um auf Speicher zuzugreifen, der **zu diesem Zeiger "gehört"**.

Kombiniert zwei Ideen: **räumliche Sicherheit** und **zeitliche Sicherheit**

Betrachtet Zeiger („pointer“) als Ressourcen: Triple (**p**, **b**, e)

- **p** ist der eigentliche Zeiger
- **b** ist die Basis des Speicherbereichs, auf den er zugreifen kann
- **e** ist die Größe (Grenze) dieses Bereichs („count“)

Zugriff ist erlaubt, wenn  $\mathbf{b} \leq \mathbf{p} \leq (\mathbf{e} - \text{sizeof}(\text{typeof}(\mathbf{p})))$

## Operationen:

Zeigerarithmetik zählt **p** hoch, lässt **b** und **e** unberührt.

Verwendung von **&**: **e** wird durch die Größe des ursprünglichen Typs bestimmt

Buffer Overflows verletzen die räumliche Sicherheit

```
void copy(char *src, char *dst, int len)
{
    int i;
    for (i=0;i<len;i++) {
        *dst = *src;
        src++;
        dst++;
    }
}
```

Das überschreiten der Grenzen von **\*dst**/**\*src** impliziert, dass **src** oder **dst** illegal sind



Beim Versuch, auf einen nicht definierten Speicherbereich zuzugreifen, wird die **zeitliche Sicherheit** verletzt.

- Räumliche Sicherheit gewährleistet, dass es sich um einen „legalen“ Bereich handelt.
- Zeitliche Sicherheit stellt sicher, dass der Bereich noch „im Spiel“ ist.

Annahme: Jeder Speicherbereich ist entweder **definiert** („defined“) oder **undefiniert** („undefined“)

- **Definiert**: Speicherbereich ist zugewiesen (und aktiv / in Nutzung)
- **Undefiniert**: Speicherbereich ist nicht zugewiesen, uninitialized oder deallokiert

Annahme: Unendlich großer Speicher, keine Wiederverwendung

Einen **Dangling-Pointer** zu verwenden (freigegebener Pointer, der jedoch vom Programm weiter verwendet wird) verletzt die zeitliche Sicherheit.

```
int *p = malloc(sizeof(int));  
*p = 5;  
free(p); // deallocate p  
printf("%d\n", *p); // violates temporal safety!
```

Der Speicherbereich gehört nicht mehr länger zu **p**.

Der Zugriff auf nicht initialisierte Zeiger ist ebenso unsicher:

```
int *p;  
*p = 5; // violates temporal safety!
```

# Beispiel: Integer Overflows

Integer Overflows an sich sind weiter erlaubt...

```
int f()
{
    unsigned short int x = 65535;
    x++;                               // overflows → x = 0
    printf("%d\n",x);                  // memory safe
    char *p = malloc(x);               // allocates a size-0 buffer
    p[1] = 'a';                        // violates spatial safety!
}
```

... jedoch keine illegalen Pointer.

Oft werden Buffer Overflows durch Integer Overflows ermöglicht!

# Memory Safety:

## Wie erreicht man Memory Safety?

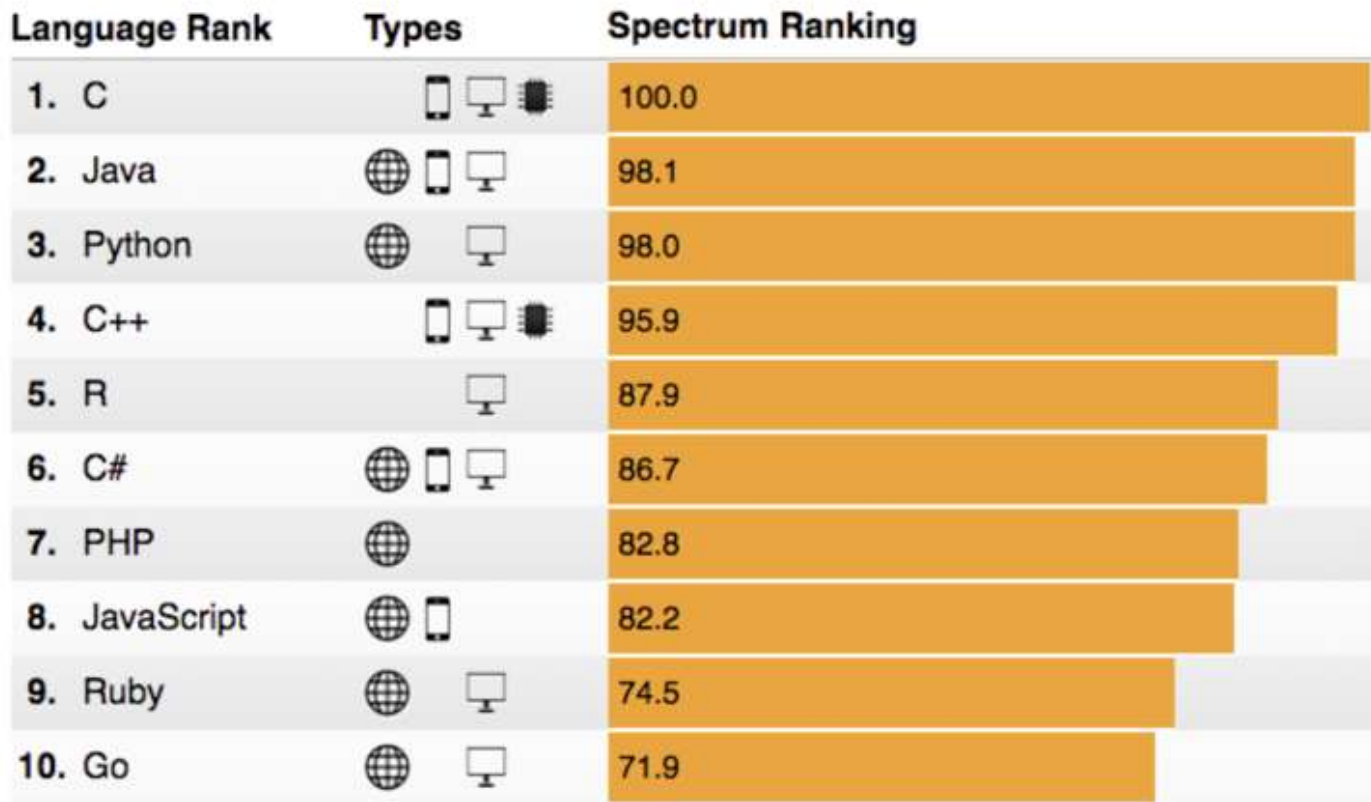
Einfachster Weg: Eine **Programmiersprache mit Memory Safety** verwenden  
(und kein C/C++)

Viele moderne Programmiersprachen sind Memory-Safe:

- Java, Ruby, Python, C#
- Haskell, Scala, Go, Rust
- usw.

Diese Sprachen sind sogar **Type-Safe**, was noch besser ist.

C / C++ sind  
noch immer  
sehr verbreitet  
(Wissenschaft)



<https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages>

C und C++ sind noch immer sehr verbreitet (und werden das auch bleiben)

⇒ Es ist nötig, auch bei diesen Sprachen Memory Safety umzusetzen.

Man **kann** damit Programme schreiben, die Memory Safe sind, doch die Sprache gibt keine Garantie.

- Compiler kann Code ergänzen, der auf Nichteinhaltung der Memory Safety prüft  
Beispiel Out-Of-Bounds: Sofortige Fehlermeldung und Programmabbruch (z.B. Java *ArrayOutOfBoundsException*)
- Idee jetzt seit über 20 Jahren bekannt
- **Limitierender Faktor: Performance**
  - Ansätze von Jones und Kelly (1997) fügen 12-fachen Overhead hinzu
  - Valgrind *Memcheck* fügt 17-fachen Overhead hinzu

# Memory Safety in C/C++: Fortschritte in der Forschung

**CCured** (2004): 1,5-fache Verlangsamung

- Keine Prüfung in Bibliotheken
- **False Positives:** Compiler weist zahlreiche sichere Programme zurück

**Softbound/CETS** (2010): 2,16-fache Verlangsamung

- Komplettes Prüfen, hohe Flexibilität

**Intel MPX Hardware** (2015 in Linux)

- Hardware-Unterstützung um die Prüfung schneller zu machen

Jedes **Objekt** hat einen bestimmten **Typ** (z.B. ein Integer, Zeiger zu einem Integer, Zeiger auf eine Funktion)

&

**Operationen** mit einem Objekt sind immer **kompatibel** mit dem Typ des Objekts

**Type Safety** ist stärker als **Memory Safety**!



# Type Safety: Wie umgehen mit C/C++?

Was können wir tun, bis wir einen weit verbreiteten typsicheren Ersatz für C/C++ haben?

Die **Ausnutzung von Fehlern erschweren!**

- Programmabsturz herbeiführen, aber keine Code-Ausführung

**Verhindern von Fehlern** durch bessere Programmierung

- Sichere Code-Praktiken, Code-Überprüfung, Review, Testen

Idealerweise beides: **Bugs verhindern und Sicherungen einbauen**, falls welche durchrutschen.

Erinnerung: Schritte für den Exploit eines Buffer Overflows

- Angreifende fügen Code in den Speicher ein (Shellcode)
- Angreifende ermitteln den **IP** Pointer zum Angriffscod
- Angreifende finden die Rücksprungadresse

Frage: Wie können wir diese Schritte komplizierter machen?

# Exploits verhindern: Overflows erkennen mit canaries

Integrität von Kohlebergwerken im 19. Jahrhundert:

- Frage: Ist meine Miene sicher? Gibt es hier giftige Gase?
- Antwort: Schicken Sie einen Kanarienvogel hinein und schauen Sie, ob er stirbt.

<https://www.smithsonianmag.com/smart-news/story-real-canary-coal-mine-180961570/>



Wir können dasselbe machen um die Integrität des Stacks zu prüfen!

# Exploits verhindern: Overflows erkennen mit canaries



- Der „Kanarienvogel“ („canary“) wird an das Ende des Buffers geschrieben und bleibt immer gleich, außer es kommt zu einem Buffer Overflow.
- Prüfung vor jeder Funktionsrückgabe: Ist der Canary unverändert?
  - Wenn nicht der erwartete Wert: Abbrechen

Frage: Welchen Wert sollte der Canary haben?

# Exploits verhindern: Overflows erkennen mit Kanarienvögeln

## Mögliche Werte für den Canary:

### 1. Terminator Canaries (z.B. CR, LF, NUL, -1)

- Nutzt die Tatsachen, dass scanf und ähnliche diese Werte nicht zulässt.
- Schwachstelle: Angreifende könnten sie in den Angriffscode einbauen

### 2. Zufällige Canaries

- Verwendung eines neuen zufälligen Canaray-Wertes bei jedem Prozessstart.
- Tatsächlicher Wert wird irgendwo im Speicher abgelegt
- Gespeicherter Wert muss schreibgeschützt sein!
- Schwachstelle: Angreifende könnten Zufallswert lesen können

### 3. Zufällige XOR Canaries

- Wie bei Zufälligen Canaries, jedoch werden stattdessen Canary XOR einige Kontrollinformationen gespeichert
- Schwachstelle: Wie Zufällige Canaries, jedoch schwerer zu lesen

- Stack Nicht-Ausführbar machen
- Randomisierung des Adressraum-Layouts
- Integrität des Kontrollflusses
- Und viele andere.  
Nicht Teil dieser Einführungsveranstaltung!

Spoiler: Nichts hilft gegen kreative Angreifende!