

Malware-Analyse und Reverse Engineering

7: Schutzmechanismen und
statische Codeanalyse

4.5.2017

Prof. Dr. Michael Engel

Teilweise basierend auf Unterlagen von Bart Coppens

<https://www.bartcoppens.be/>



Überblick

Themen:

- Schutzmechanismen gegen Buffer Overflows und ROP
- Statische Codeanalyse: Suche nach Security-Bugs

Schutzmechanismen

Softwaremethoden

- Stack smash protector / stack canaries
- Shadow stacks

Hardwaremethoden

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP, W^X)

Stack Canaries

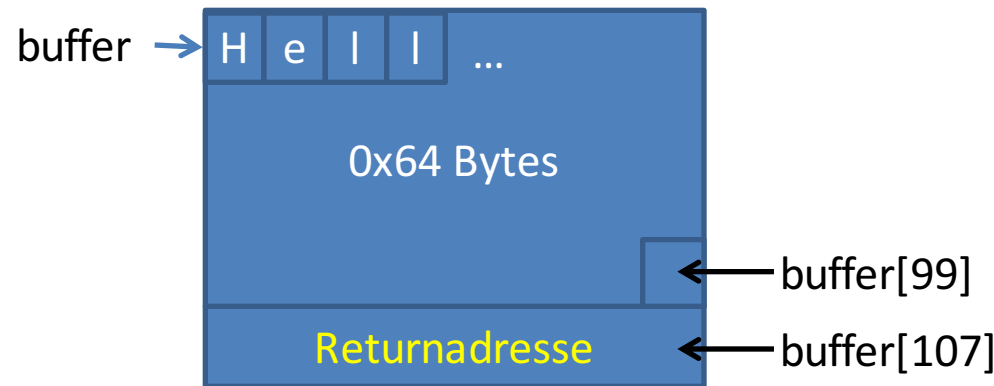
Ziel: Erkennen von Buffer overflows

- Idee: Platzieren eines besonderen Wertes („canary“) zwischen lokalen Variablen und Returnadresse
 - gcc: „-fstack-protector“
- Setzen bei Eintritt in die Funktion
- Vor Return aus Funktion:
Überprüfen, ob Wert noch unverändert ist
- Buffer overflow muss canary überschreiben, um die Returnadresse manipulieren zu können



Stack Canary/Cookie

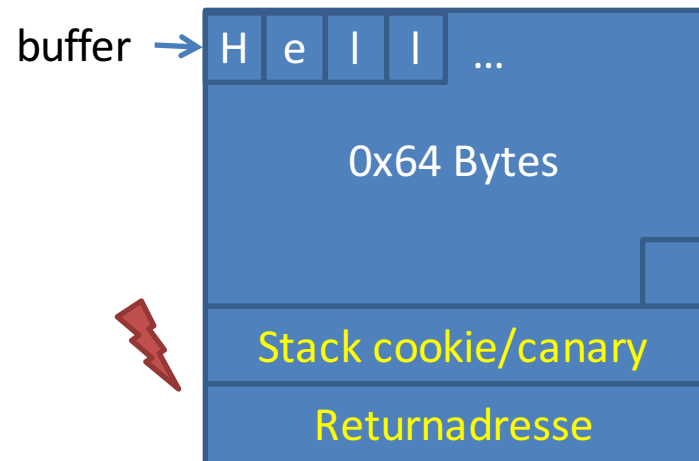
Zustand **ohne** Stack canary:
Schreiben über buffer[99] hinaus
überschreibt die Returnadresse



```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
add    $0x64, %rsp
retq
```

Stack Smash Protector / Cookies

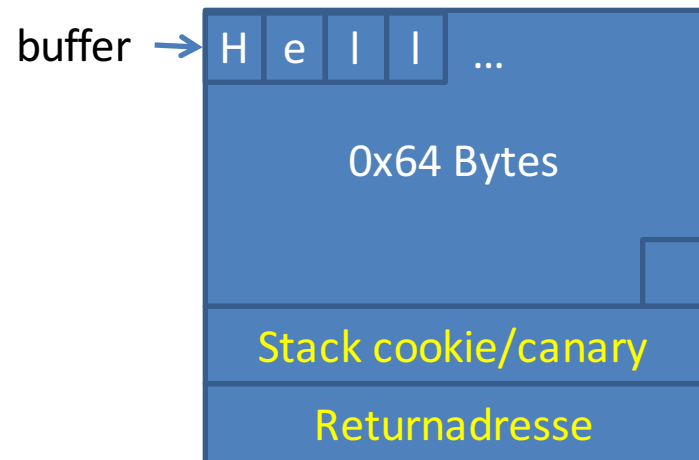
Idee: Einfügen eines Schutzes zwischen buffer und Returnadresse
⇒ benötigt Codeanpassung, da Schutz an Stelle der Returnadresse liegt



```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
add    $0x64, %rsp
retq
```

Stack Smash Protector / Cookies

Codeanpassung, da Schutz an Stelle der Returnadresse liegt
⇒ zusätzliche Bytes auf dem Stack reservieren (0x6C statt 0x64)



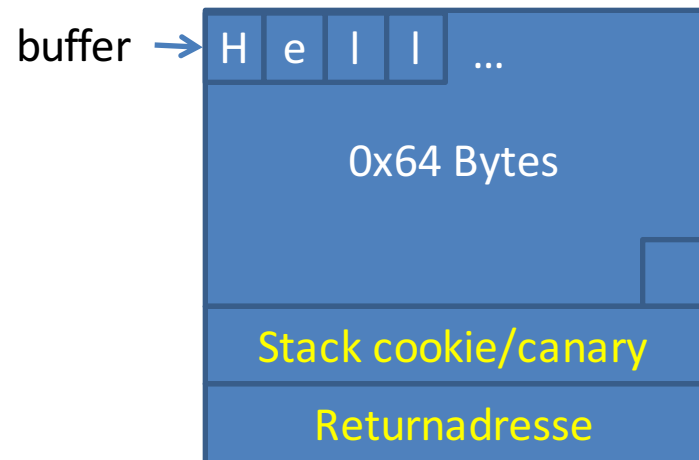
```
sub    $0x6C, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
add    $0x6C, %rsp
retq
```

Stack Smash Protector / Cookies

Einfügen des Stack Canaries:

Hier: Per Prozess eindeutiger Wert, von Loader (dyld) erzeugt.

Segment-relative Adressierung → Wert schwer zu ermitteln

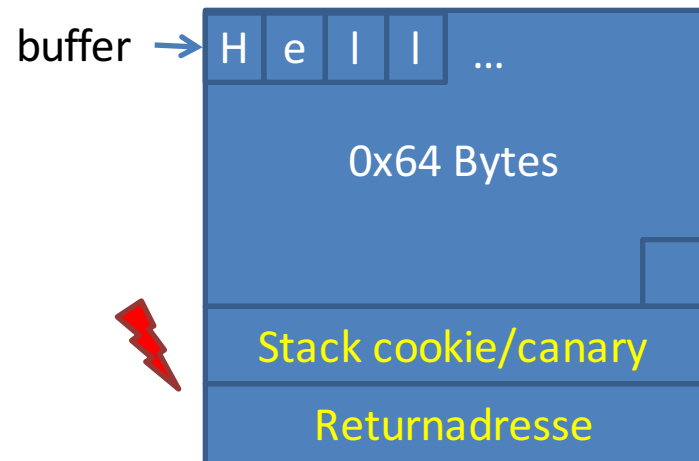


```
sub    $0x6C, %rsp
mov     %fs:0x28,%rax
mov     %rax,0x64(%rsp)
movq    %rdi,%rdx
movq    %rsp,%rdi
callq   <memcpy>
add     $0x6C, %rsp
retq
```

Adressen →

Stack Smash Protector / Cookies

Eingefügter Code zum **Überprüfen des Stack Canaries**:
Wenn nicht identisch mit dem am Anfang der Funktion
geschriebenen Wert → Abbruch

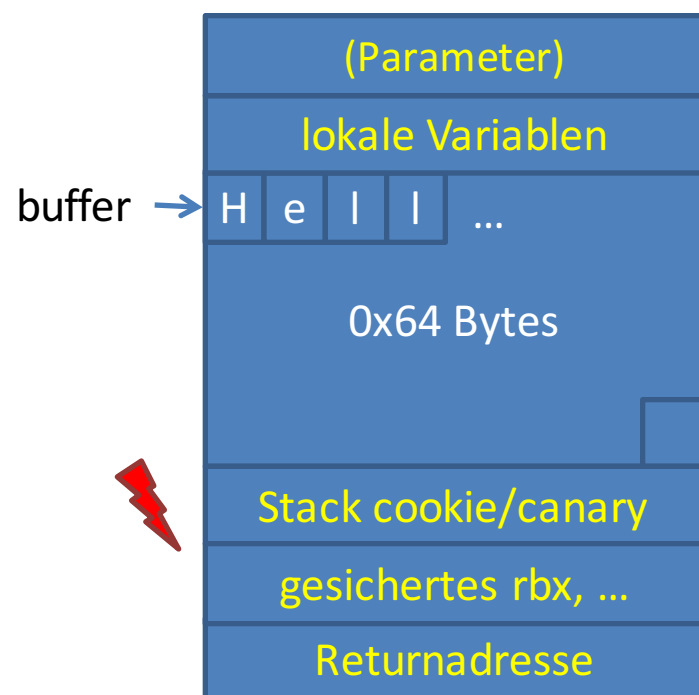


```
sub    $0x6C, %rsp
mov    %fs:0x28,%rax
mov    %rax,0x64(%rsp)
movq   %rdi,%rdx
movq   %rsp,%rdi
callq  <memcpy>
mov    0x64(%rsp),%rax
xor    %fs:0x28,%rax
jne    ... stack_chk_fail
add    $0x6C, %rsp
retq
```

Adressen
↓

Stack Smash Protector / Cookies

Funktioniert auch, wenn Basepointer rbx vor Return-
adresse auf Stack liegt → wird ebenfalls geschützt



```

sub    $0x6C, %rsp
mov    %fs:0x28,%rax
mov    %rax,0x64(%rsp)
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
mov    0x64(%rsp),%rax
xor    %fs:0x28,%rax
jne    ... stack_chk_fail
add    $0x6C, %rsp
retq

```

Werte für Canaries

Fester Wert

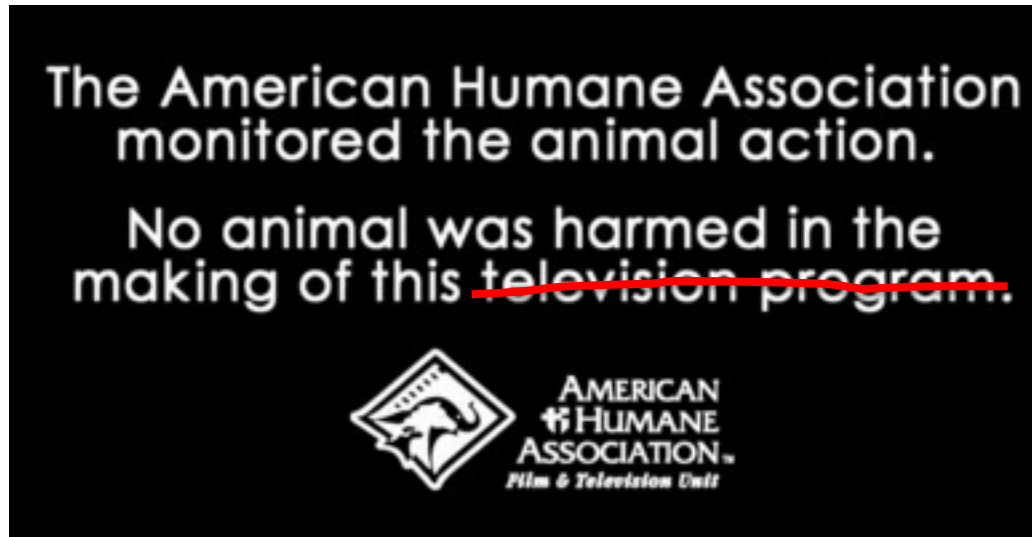
- Idee: Wahl eines speziellen Wertes für das Canary, der mit gets, scanf, strcpy so nicht auf dem Stack erzeugt werden kann
 - sog. „Terminator Canary“
- 4 Bytes: **0xff0a0d00** – 0, CR, LF, -1 (LSB->MSB)
 - CR oder LF ist Endekennzeichen für gets/scanf
⇒ brechen beim Versuch der Übergabe in String ab
 - 0x00 ist Endekennzeichen für strcpy und verwandte Funktionen

Zufällige Werte

- Zufällige Bytes zur Ladezeit von dyn. Loader erzeugt
- In geschütztem Speicherbereich
- Guter Zufall erforderlich, sonst brute force-Angriff möglich!

Warum “Canaries”?

Wikipedia: “the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system.”



lecture

Sind Canaries sicher?

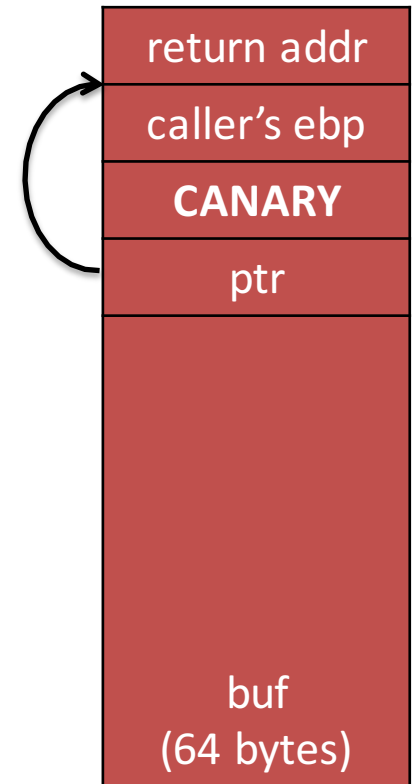
Angriffe auf Canaries

- Nutzen speziellere Konstellationen von lokalen Variablen aus $\Rightarrow$ schwieriger auszunutzen, aber nicht unmöglich!

Zuerst einen Datenzeiger überschreiben

- „Data Pointer Subterfuge“:

```
int foo(void) {  
    int *ptr  
    char buf[64];  
    memcpy(buf, user1);  
    *ptr = user2;  
    ...  
}
```

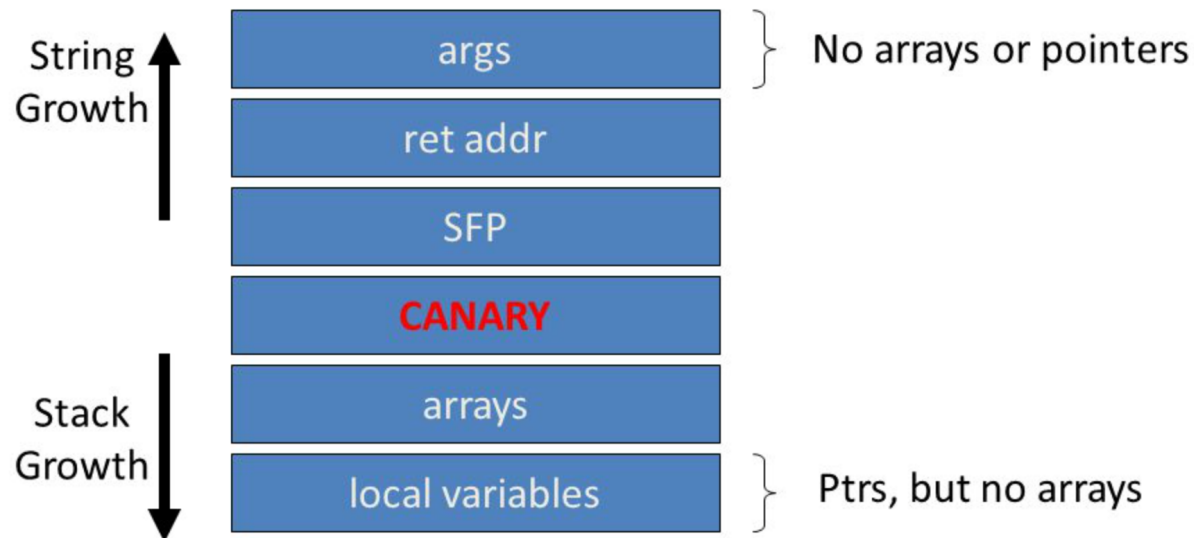


[https://msdn.microsoft.com/en-us/library/aa290051\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx)

Verhindern des Überschreiben von Pointern

gcc „ProPolice“

- Wenn möglich, den Stack so umorganisieren, dass Arrays über allen anderen lokalen Variablen liegen
- Teil von „-fstack-protector“



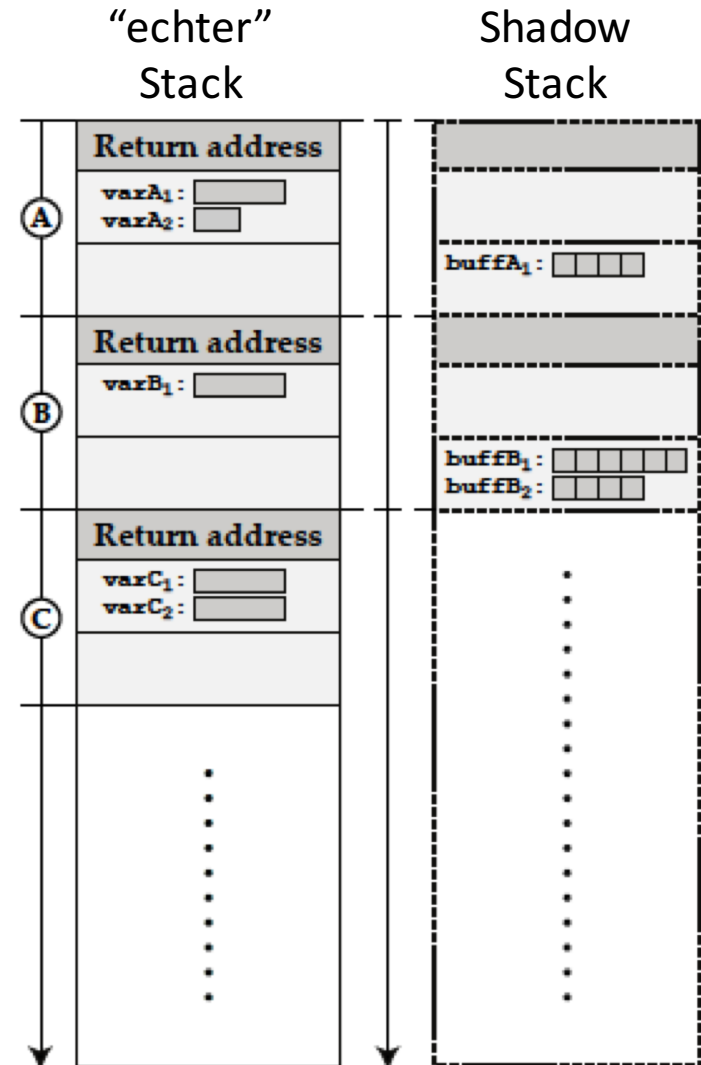
Shadow Stacks

Ziel: Vermeiden des Ausnutzens von Buffer overflows

- Idee: Aufteilen des Stacks in zwei separate Bereiche:
 - Lokale Variable usw.
 - Returnadressen
- Returnadressen in separatem Bereich
⇒ nicht von Buffer Overflow überschreibbar

Shadow Stacks

- **Zwei Stacks** in separaten Speicherbereichen
- Format der Stackframes ist in beiden Bereichen identisch
 - Keine aufwendige Offset-Umrechnung notwendig
- Der erste enthält normale Variablen und die Return-adressen: adressiert durch `rsp`
- Der zweite enthält nur potentiell überlaufende Buffer
 - Adressiert durch ein anderes Register



Shadow Stacks

- **Problem:**
höherer Aufwand erforderlich für push/pop
- Hier: Verwendung von `ebp` als „Stackpointer“ für den shadow stack
 - Damit wird normaler Stack ohne `ebp` adressiert...

“echter” Stack (control stack):

```
// push instruction
push %rbx

// pop instruction
pop %rbx
```

Shadow Stack:

```
// push simulation
sub 8, %rbp
mov %rbx, (%rbp)
// pop simulation
mov (%rbp), %rbx
add 8, %rbp
```

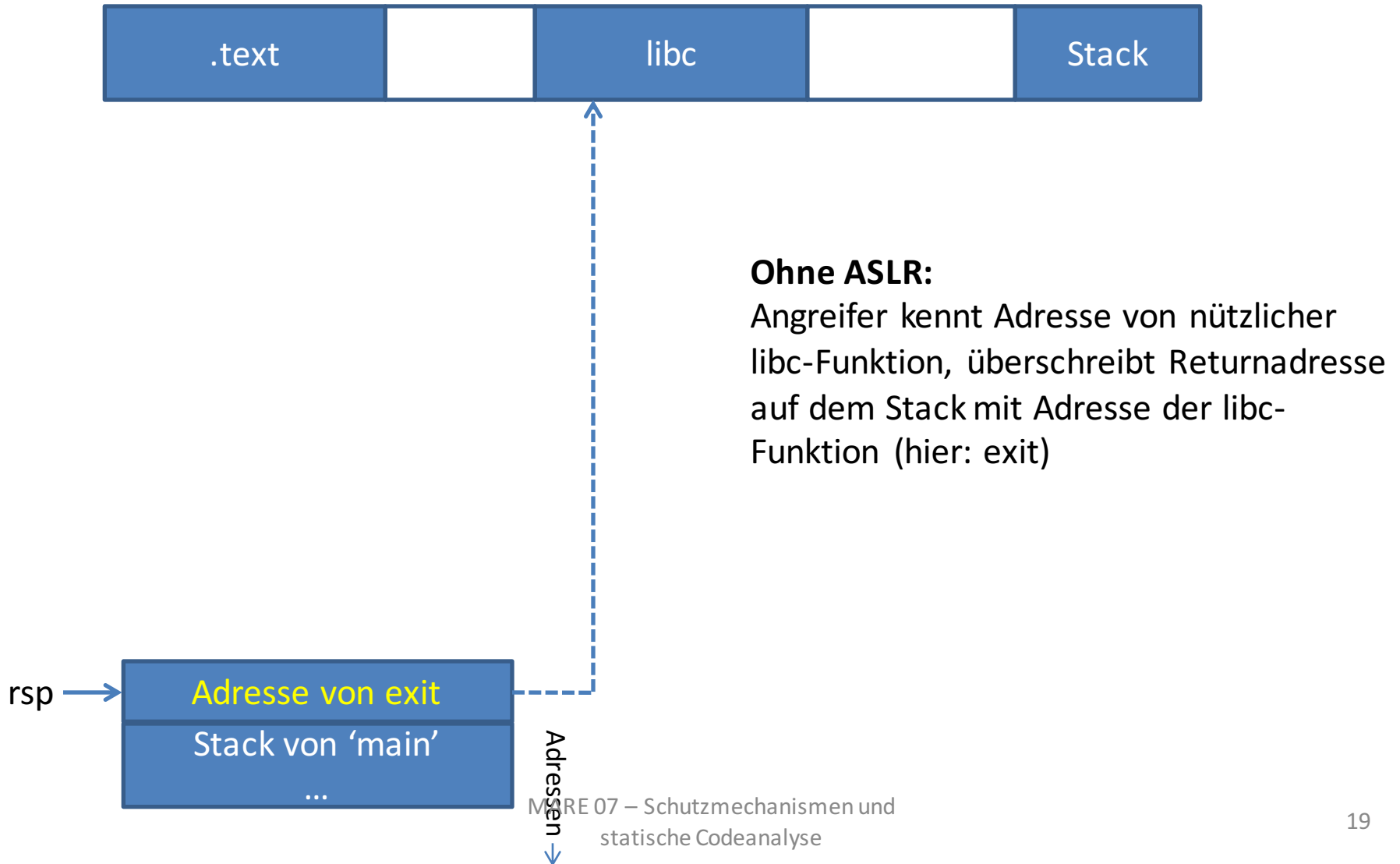
Christopher Kugler, Tilo Müller: *“SCADS: Separated Control- and Data-Stacks”*
(Best Student Paper Award), 10th International Conference on Security and Privacy
in Communication Networks (SecureComm '14) – Paper aus Erlangen

Address Space Layout Randomization (ASLR)

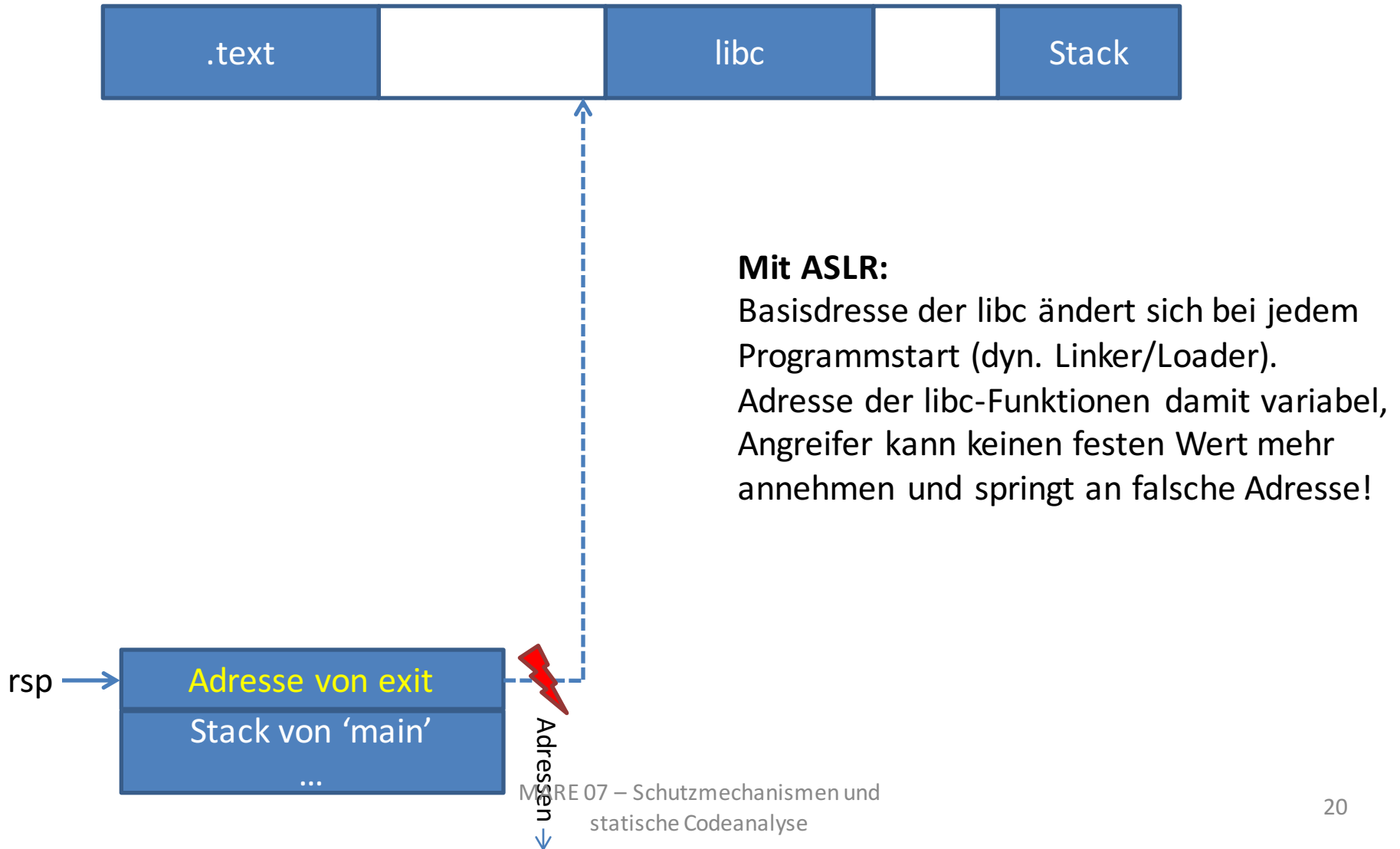
Ziel: Aufrufen von Library-Funktionen erschweren

- Malware nutzt Wissen über bekannte Adressen (Funktionszeiger) von libc-Funktionen aus
- Idee: Zufällige Wahl von Offsets von libc-Funktionen beim Laden des Programms
- Folge: Angreifer hat es schwer, die Returnadresse auf dem Stack mit einer festen Adresse einer libc-Funktion zu überschreiben
 - Statt Funktionsaufruf: Absturz bei Verwendung von ASLR

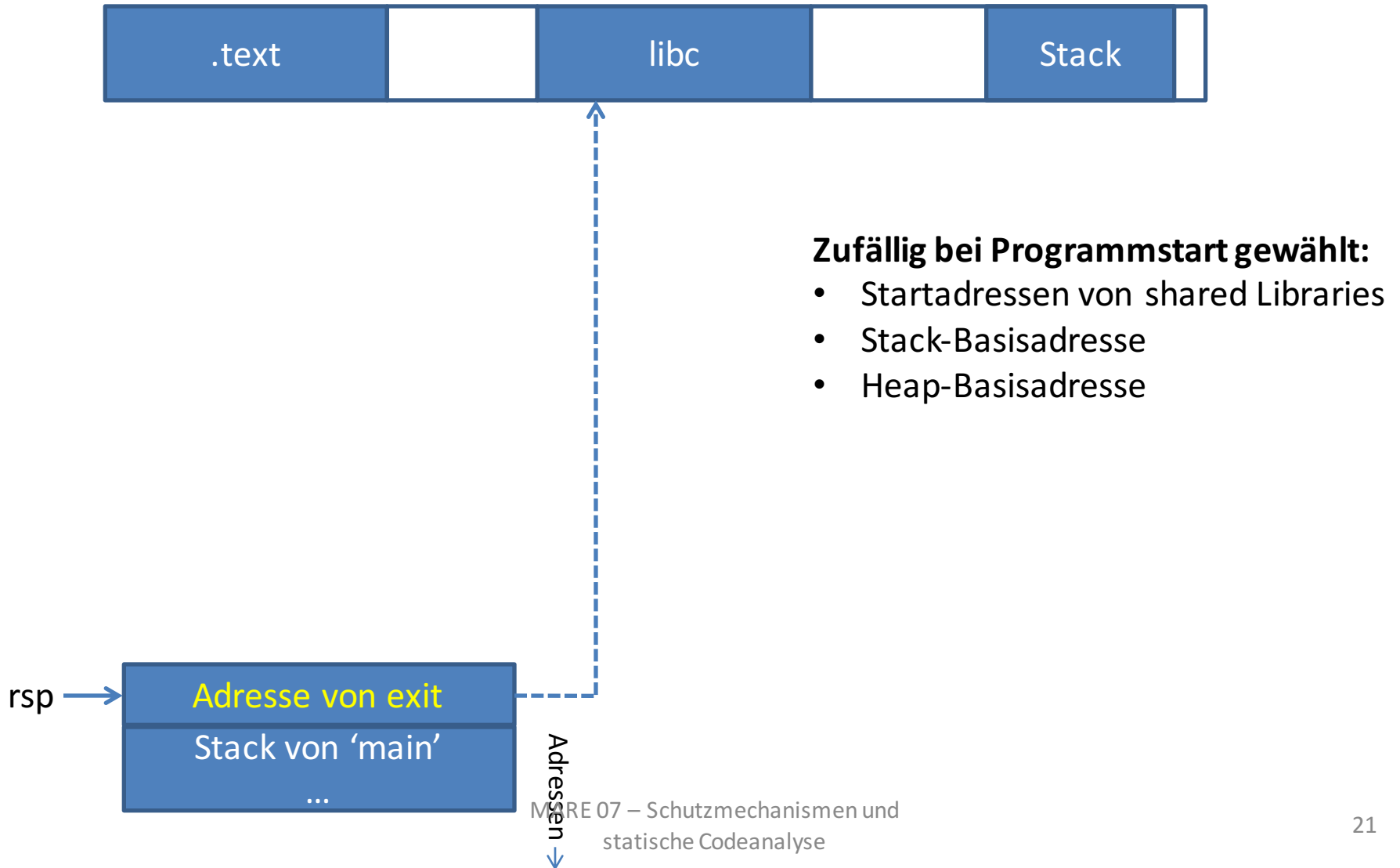
Address Space Layout Randomization



Address Space Layout Randomization



Address Space Layout Randomization

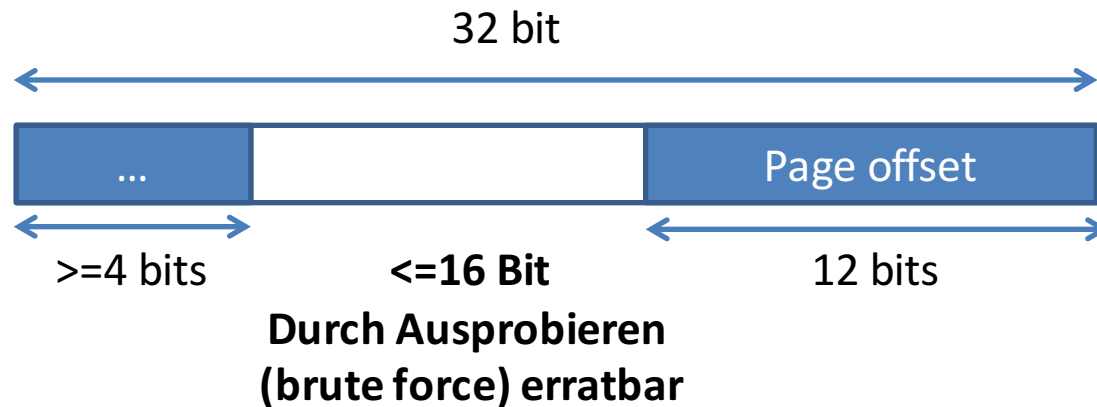


Address Space Layout Randomization



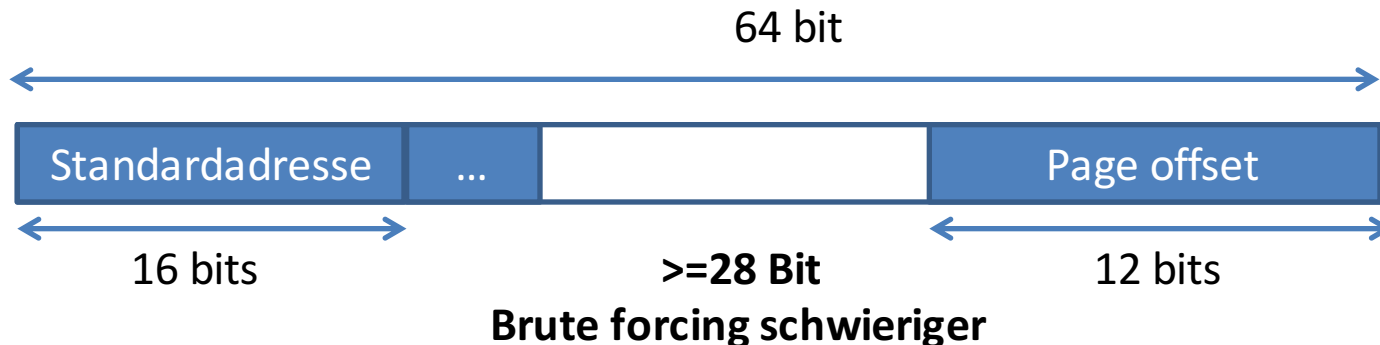
32 Bit-Adressen:

- Wenig Spielraum für zufällige Adressen
- Kann "geraten" werden

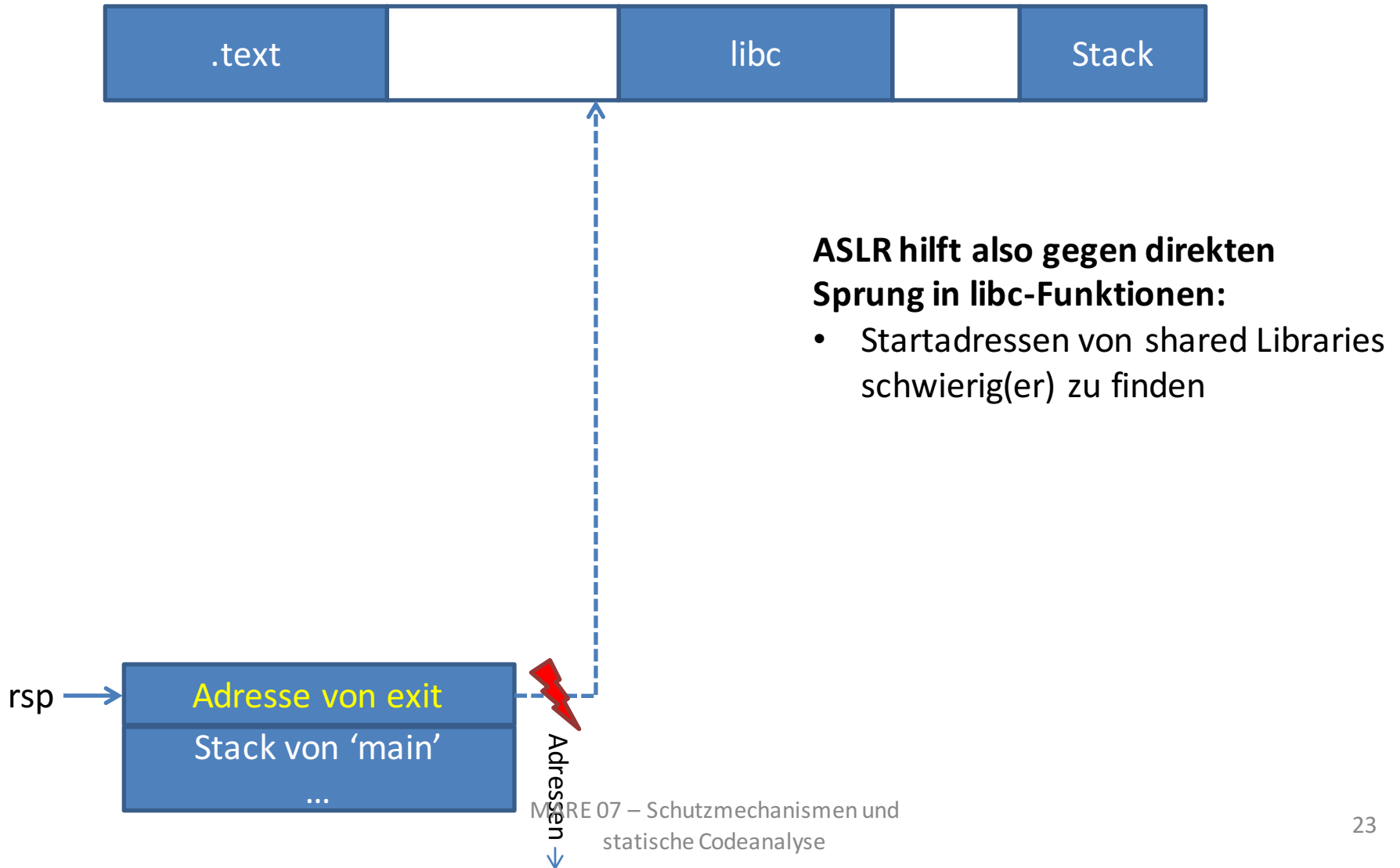


64 Bit-Adressen:

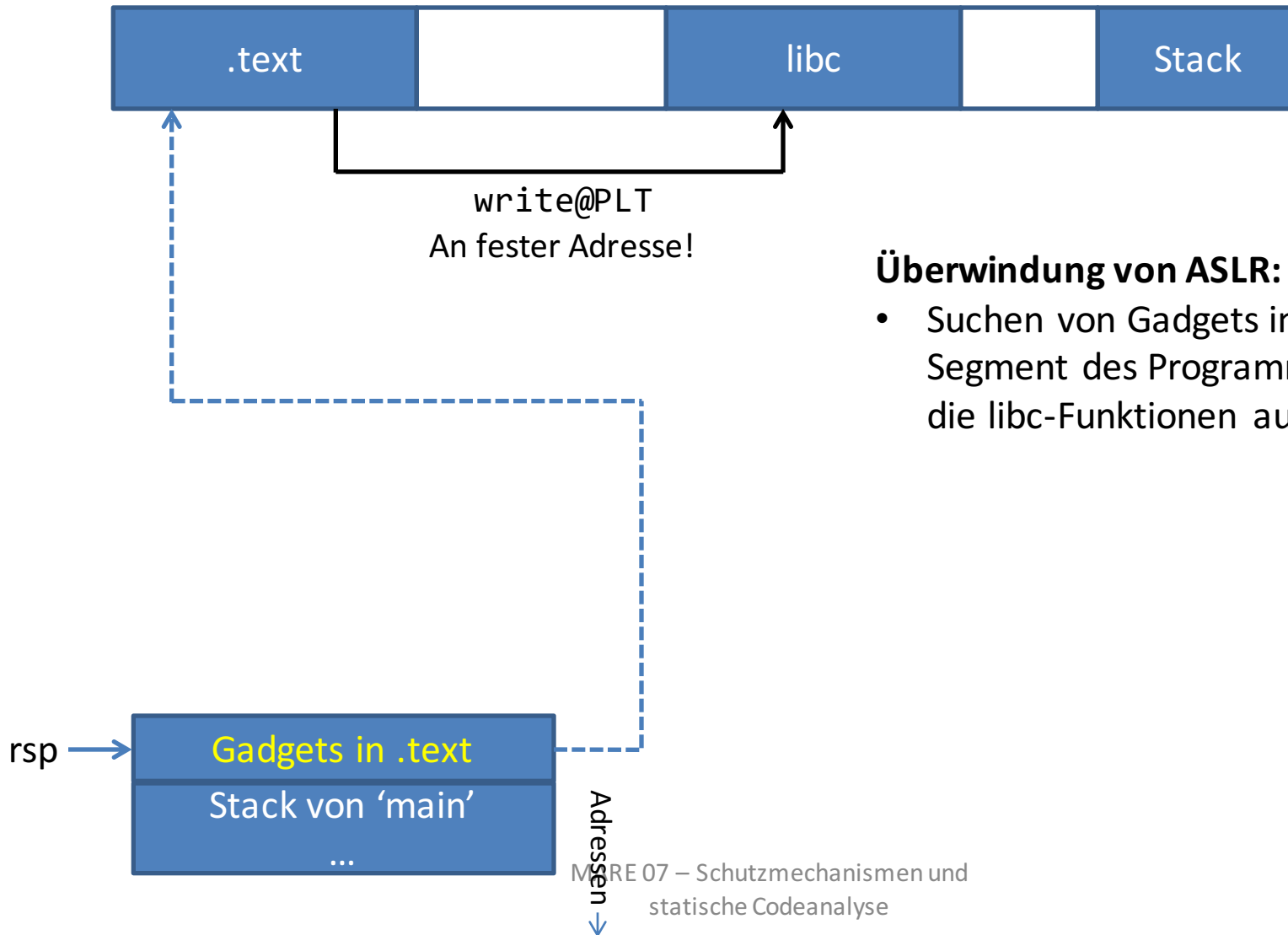
- Spielraum ist 2^{12} -mal größer
- Aufwendig zu erraten



Address Space Layout Randomization



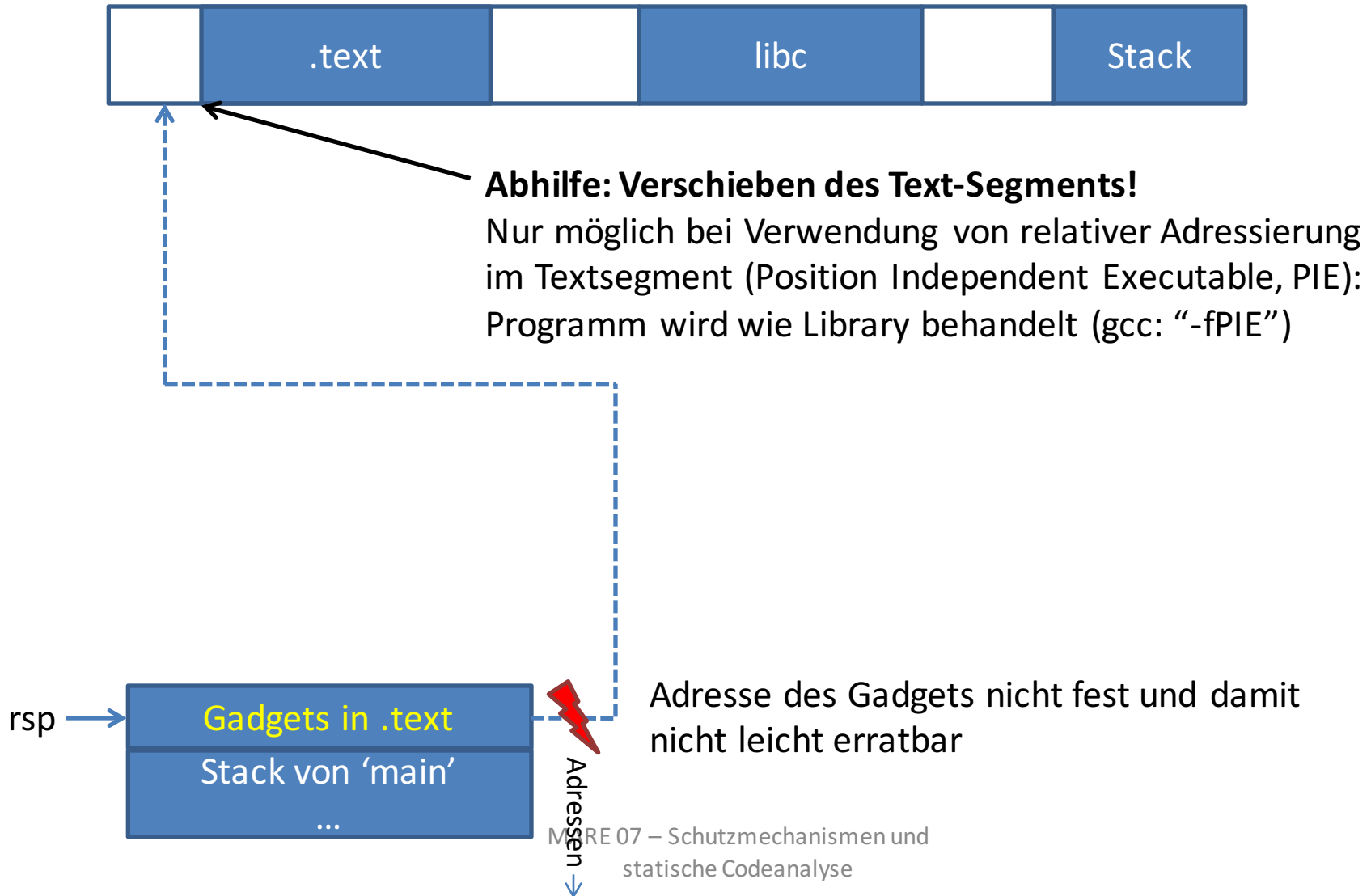
Address Space Layout Randomization



Überwindung von ASLR:

- Suchen von Gadgets im Text-Segment des Programms selber, die libc-Funktionen aufrufen!

Address Space Layout Randomization



Data Execution Prevention

Ausführbarer Stack

- Code kann direkt auf dem Stack abgelegt werden
- „Mitliefern“ von Code für Malware-Funktionalität

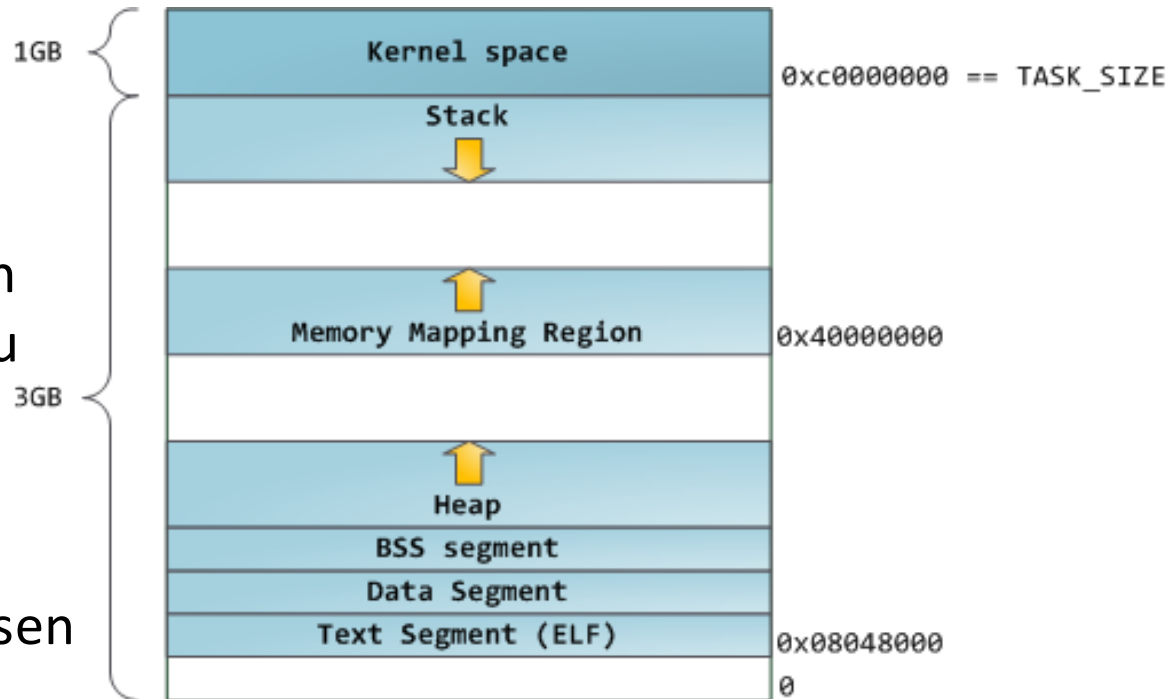
Abwehrmaßnahme

- Abschalten der Möglichkeit, Code auf beschreibbaren Speicherseiten auszuführen (in Hardware)
- Auch „W^X“ genannt (Write XOR Execute)

Speicherlayout von x86

Hier: 32 Bit Linux

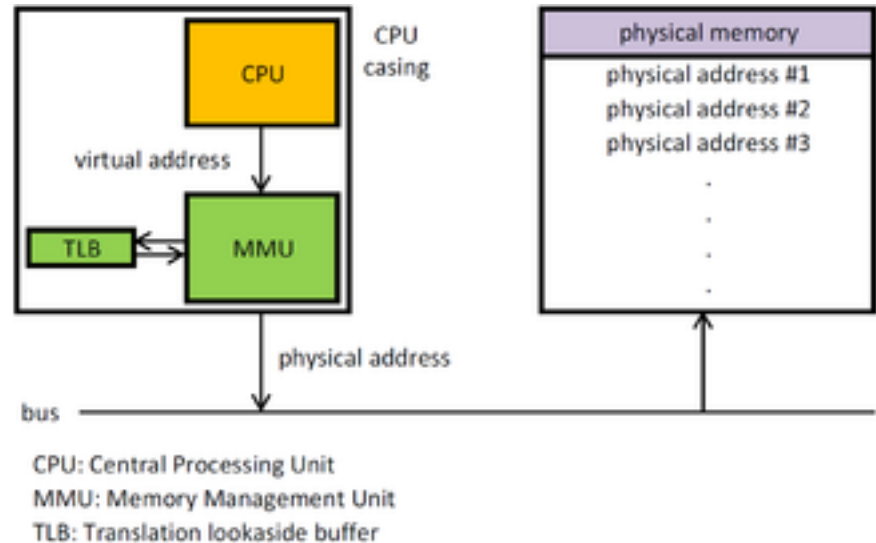
- Jeder Prozess hat Illusion, den gesamten Adressraum für sich zu besitzen (0-3 GB)
- Konflikt, wenn > 1 Prozess läuft...
- Lösung: Prozessadressen werden *virtualisiert*
- Übersetzung zu „echten“ physikalischen Adressen im Speicher
 - Speicher wird unter Prozessen aufgeteilt
 - Hardware-Komponente: MMU
 - Granularität: „Seite“ (page) von 4 kB (auch größere möglich)



MMU: Memory Management Unit

Übersetzung von virtuellen zu physikalischen Adressen

- Sitzt zwischen der CPU und dem Hauptspeicher
- MMU heute Teil der CPU
- CPU erzeugt virtuelle Adressen
- MMU setzt diese über Seitentabelle in physikalische Adressen um und übergibt diese an den Speicher (Seitentabelle liegt im RAM!)
- Optimierung: „Translation Lookaside Buffer“ (TLB)
 - Kleiner Cache für Seitentabellen-(Übersetzungs-)einträge



Adressumsetzung bei x86

MMU (Memory Management Unit) der CPU übersetzt mit Hilfe der Seitentabelle virtuelle in physikalische Adressen

- Einheit: „Seite“ (page), 4 kB
- Virtuelle Adresse zerlegt in mehrere Teile
- Teile sind Index in Seitentabelle
 - Diese ist hierarchisch aufgebaut

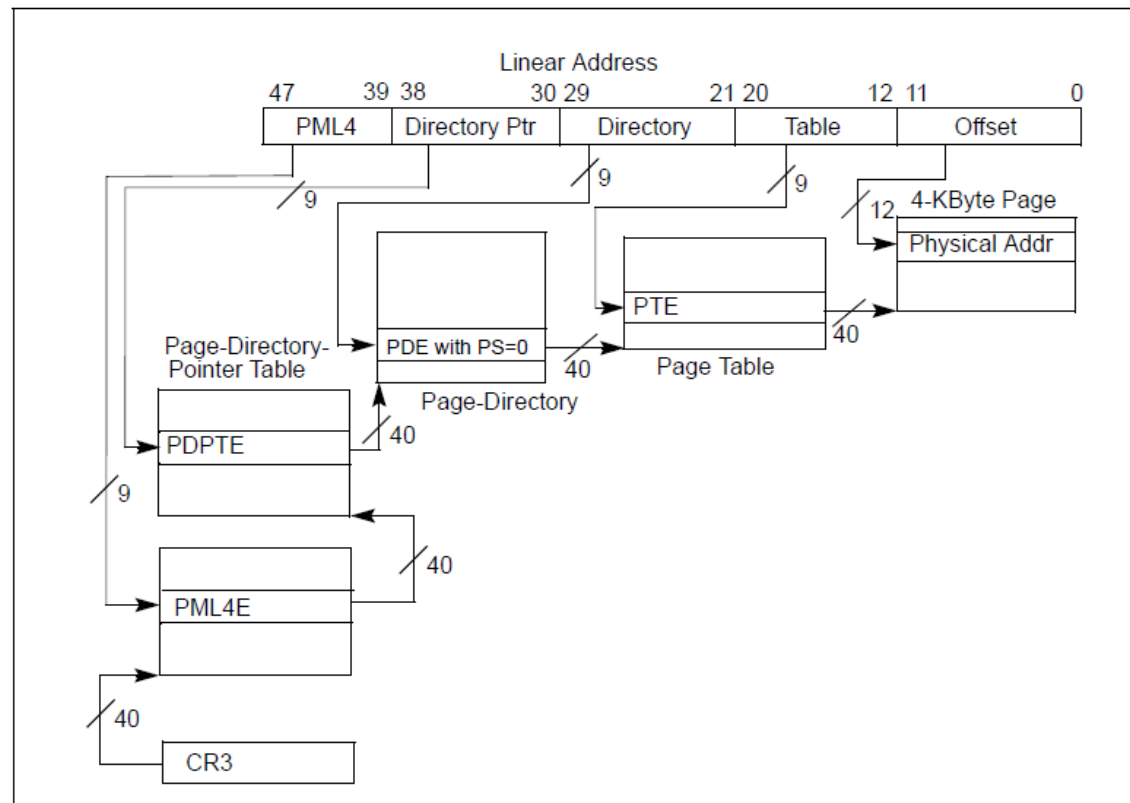


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

- Enthalten phys. Page-Nummer zu log. Page-Nr.
- zusätzlich Metainfos
- Schutzbit „XD“: Execute Disable

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Weitere ausnutzbare Eigenschaften

Buffer overflows sind größte Gruppe (ca. 50% der Sicherheitslücken), aber viele weitere Probleme:

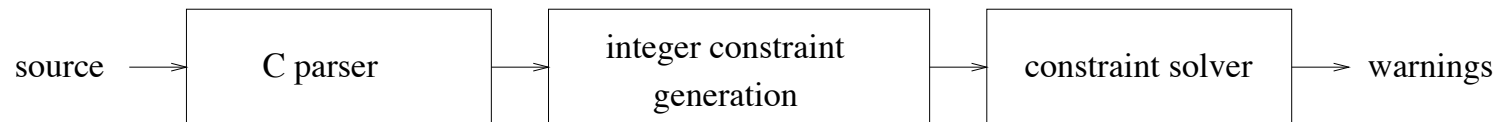
- Ausnutzung von doppeltem free/use after free usw.
- Heap spray, heap grooming
- Typ-Konfusion
- Race Conditions
- Integer-Überläufe
- Entsprechende Probleme im Kernel, nicht im Usermode
- und vieles anderes mehr...
- Beispiele bei googleprojectzero.blogspot.com
- Literatur: "The Art of Software Security Assessment"

Statische Codeanalyse

Idee

- Mögliche Sicherheitsprobleme bereits bei der Übersetzung finden und als Fehler markieren
- Besonderer Fokus: Arrayindizes
- Verwendung eines Compilers, der Arraygrenzen überprüft
 - Also Sicherstellen, dass immer genug Speicher vorhanden ist, damit Arrayeintrag mit größtem verwendeten Index „sicher“ gespeichert werden kann
 - Hat erheblichen Performance-Overhead zur Folge!

Erste statische Analysen



Spezieller „Compiler“ für C-Code

- Analysiert Code auf Buffer-Operationen (strcpy, gets, ...)
- Versucht, Informationen über maximale Längen von erzeugten Strings zu ermitteln
- Beschreibung über domänenspezifische Sprache, die Bedingungen (constraints) beschreiben kann

Erste statische Analysen: Modellierung von Strings

C-Strings werden als *abstrakter Datentyp* betrachtet

- Auf diesem werden dann Operationen wie strcpy, strcat, ... ausgeführt
- Vereinfacht die Analyse, sonst wäre allgemeine Pointeranalyse erforderlich!

Buffer werden als Paare von Integer-Wertebereichen modelliert

- Keine Analyse der *Inhalte* von Strings!
- Strings charakterisiert durch zwei Integer-Werte:
 - Anzahl an Bytes, die für den String alloziert wurden
 - Anzahl aktuell vom String verwendeter Bytes
- Standard C-String-Funktionen ändern diese Werte

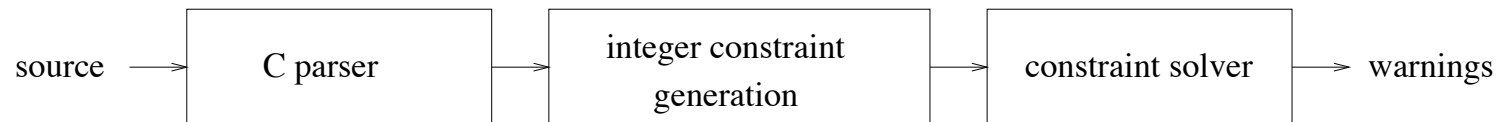
Erste statische Analysen: Constraint-Definitionen

Erkennung von Buffer overflows

- Überwachung der Integer-Bereiche
- Prüfen, ob $\text{len}(s) \leq \text{alloc}(s)$ for all string variables s .
allozierte Größe *immer* $\geq$ maximal verwendete Größe

Trennung in zwei Schritte

- Modellierung von String-Operationen als constraints auf Integer-Wertebereiche
- Lösen des constraint-Problems (Ungleichungssystem)



Erste statische Analysen: Modellierung von String-Operationen

C code	Interpretation
<code>char s[n];</code>	$n \subseteq \text{alloc}(s)$
<code>strlen(s)</code>	$\text{len}(s) - 1$
<code>strcpy(dst,src);</code>	$\text{len}(\text{src}) \subseteq \text{len}(\text{dst})$
<code>strncpy(dst,src,n);</code>	$\min(\text{len}(\text{src}), n) \subseteq \text{len}(\text{dst})$
<code>s = "foo";</code>	$4 \subseteq \text{len}(s), \quad 4 \subseteq \text{alloc}(s)$
<code>p = malloc(n);</code>	$n \subseteq \text{alloc}(p)$
<code>p = strdup(s);</code>	$\text{len}(s) \subseteq \text{len}(p), \quad \text{alloc}(s) \subseteq \text{alloc}(p)$
<code>strcat(s,suffix);</code>	$\text{len}(s) + \text{len}(\text{suffix}) - 1 \subseteq \text{len}(s)$
<code>strncat(s,suffix,n);</code>	$\text{len}(s) + \min(\text{len}(\text{suffix}) - 1, n) \subseteq \text{len}(s)$
<code>p = getenv(...);</code>	$[1, \infty] \subseteq \text{len}(p), \quad [1, \infty] \subseteq \text{alloc}(p)$
<code>gets(s);</code>	$[1, \infty] \subseteq \text{len}(s)$
<code>fgets(s,n,...);</code>	$[1, n] \subseteq \text{len}(s)$
<code>sprintf(dst,"%s",src);</code>	$\text{len}(\text{src}) \subseteq \text{len}(\text{dst})$
<code>sprintf(dst,"%d",n);</code>	$[1, 20] \subseteq \text{len}(\text{dst})$
<code>snprintf(dst,n,"%s",src);</code>	$\min(\text{len}(\text{src}), n) \subseteq \text{len}(\text{dst})$
<code>p[n] = '\0';</code>	$\min(\text{len}(p), n + 1) \subseteq \text{len}(p)$
<code>p = strchr(s,c);</code>	$p = s+n; \quad [0, \text{len}(s)] \subseteq n$
<code>h = gethostbyname(...);</code>	$[1, \infty] \subseteq \text{len}(h \rightarrow h_name),$ $[-\infty, \infty] \subseteq h \rightarrow h_length$

Da ist die Funktion,
die Ärger bereitet:
**Stringlänge im
Intervall $[1, \infty]$!**

(Ein) Problem der statischen Analyse

Buffer overflow wird nicht gefunden!

- Hier findet *Pointer aliasing* statt!
- Pointer p zeigt in den Speicherbereich des Strings, der von Pointer s adressiert wird
- Constraints interpretieren das Statement der Form $q = p + j$; als $\text{alloc}(p) - j \subseteq \text{alloc}(q)$, $\text{len}(p) - j \subseteq \text{len}(q)$
- Funktioniert, so lange auf *p und *q nicht geschrieben wird
 - Da p und q überlappen, wird Schreiben in einen String nicht auch als Schreiben in den aliasten String angesehen

```
char s[20], *p, t[10];  
strcpy(s, "Hello");  
p = s + 5;  
strcpy(p, " world!");  
strcpy(t, s);
```

David Wagner et al., “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities”, NDSS 2000

Lösen der Constraints

Algorithmus zum Lösen

- Programm mit k Variablen
- Zustandsraum $\mathbb{Z}^k$: i -tes Element enthält Wert der i -ten Variablen
- Programmausführung: Pfad durch den Zustandsraum
- Ziel: Finden eines minimalen Intervalls, das alle möglichen Pfade durch den Zustandsraum beinhaltet...

Diesen Algorithmus muss man hier nicht verstehen (und kann es aus der kurzen Beschreibung auch nicht), er dient der Darstellung, wie komplex die Analyseaufgabe ist. Bei Interesse: Details im Paper...

David Wagner et al., “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities“, NDSS 2000

CONSTRAINT-SOLVER

1. Set $\alpha(v_i) := \emptyset$ for all i , and set $done := false$.
2. For each constraint of the form $n \subseteq w$, do
3. Set $\alpha(w) := \text{RANGE-CLOSURE}(\alpha(w) \cup \{n\})$.
4. While $done \neq true$, call ONE-ITERATION.

ONE-ITERATION

1. Set $C(v_i) := \text{white}$ for all i and set $done := true$.
2. For each variable v , do
3. If $C(v) = \text{white}$, do
4. Set $\text{prev}(v) := \text{null}$ and call $\text{VISIT}(v)$.

VISIT(v)

1. Set $C(v) := \text{gray}$.
2. For each constraint of the form $f(v) \subseteq w$, do
3. If $f(\alpha(v)) \not\subseteq \alpha(w)$, do
4. Set $\alpha(w) := \text{RANGE-CLOSURE}(\alpha(w) \cup f(\alpha(v)))$.
5. Set $done := false$.
6. If $C(w) = \text{gray}$, call $\text{HANDLE-CYCLE}(v, w, \text{prev})$.
7. If $C(w) = \text{white}$, do
8. Set $\text{prev}(w) := v$ and call $\text{VISIT}(w)$.
9. Set $C(v) := \text{black}$.

RANGE-CLOSURE(S)

1. Return the range $[\inf S, \sup S]$.

Fazit

Schutzmechanismen gegen Buffer overflows

- Hard- und Softwaremethoden
- Gemeinsame Eigenschaft:
 - Erschweren von Angriffen
 - Aber weiterhin Angriffe mit größerem Aufwand möglich

Statische Analyse von Buffer overflows

- Aufwendige Analyseaufgabe, constraint solving
 - Komplexe Theorie und komplexe Algorithmen
 - Zeitaufwendig
- Probleme mit Spezialfällen, ibs. Pointer aliasing
 - Kommen so in echten C-Programmen leider vor...
- Bessere Analysen im Buch von Axel Simon (siehe Literatur)

Literatur

Stack Canaries

- Perry Wagle, Crispin Cowan, “*StackGuard: Simple Stack Smash Protection for GCC*”, GCC Summit 2003: <ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf>
- Thurston H.Y. Dang et al, “*The Performance Cost of Shadow Stacks and Stack Canaries*”, In Proc. ACM ICCS, 2015

ASLR

- H. Shacham et al., “*On the effectiveness of address-space randomization*”, Proceedings of the 11th ACM Conference on Computer and Communications Security, 2004

Weitere Sicherheitslücken

- Mark Dowd et al., “*The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*”, Addison-Wesley 2006, ISBN-13: 978-0321444424

Statische Codeanalyse

- Axel Simon, “*Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*”, Springer 2011, ISBN-13: 978-1-84800-016-2