Complex Lab “Systems Programming”
— Day 6 —
Debugging and Assembly

Jan Bierbaum
Maksym Planeta
Björn Döbel
Tobias Stumpf

2023-09-26
Prologue: Evaluation

Please participate and help to improve the lab

https://befragung.zqa.tu-dresden.de/uz/de/sl/T7f6W2wfEFp7
Some Ethymology/History

Rear Admiral Grace Murray Hopper

1947: “First actual case of bug being found”
Some Ethymology/History

Rear Admiral Grace Murray Hopper

1947: “First actual case of bug being found”
Definitions

Bug . . . flaw in a computer system that results in unexpected behavior
Debugging . . . process of searching and fixing deviations from the expected behavior
Debugging is not only finding living creatures in an electronic device:

- Programme crash
- Wrong result
- Slow execution
How to debug?
How to debug? Example: Digging
How to debug? Example: Digging
How to debug? Example: Digging
How to debug? Example: Digging
How to debug? Example: Digging
Debugging Tools

- strace
- ltrace
- gdb
- valgrind
- perf
- ptrace
- and even more...
Tracing System Calls — strace

Inspect system calls performed by a programme

- Filtering: `strace -e`
- Timing: `strace -t[tt] / strace -T`
- Statistics: `strace -c`
Assignment №1

1. Which system calls are performed when you run `/bin/ls`?
2. How many calls are performed?
3. Why so many?
Tracing library calls — ltrace

Inspect all calls to *shared* libraries

- Filtering: `ltrace -e`
- Timing: `ltrace -t[tt] / ltrace -T`
- Statistics: `ltrace -c`
Assignment №2

$ wget https://os.inf.tu-dresden.de/Studium/SysProg/SS2023/debugging/strace.tar.xz

$ tar -xJf strace.tar.xz

$ cd strace

Make it print “SUCCESS”!

Hints: file, strace / ltrace
Problem: Memory Leaks

1. Allocate memory buffer
2. Use the buffer
3. Stop using the buffer
4. (Optional) Loose pointer to the buffer
5. Rinse and repeat
Dynamic Linker

- Recall static linking vs. dynamic linking

Details: man ld.so
Dynamic Linker

- Recall static linking vs. dynamic linking
- Resolves symbols by searching for libraries in \texttt{LD\_LIBRARY\_PATH}

Details: \texttt{man ld.so}
Dynamic Linker

- Recall static linking vs. dynamic linking
- Resolves symbols by searching for libraries in LD_LIBRARY_PATH
- LD_PRELOAD
  - Force loading of libraries
  - Loaded before any other dynamic library
  - Application has no choice

Details: man ld.so
Detecting Memory Leaks

• Use LD_PRELOAD to let the leaky programme call custom implementations of malloc/free
• Track malloc/free information to report memory leaks at programme termination
• Use the real malloc/free to perform the actual work
Interfacing with the Dynamic Linker

```c
void* dlopen(const char* filename, int flag);
char* dlerror(void);
void* dlsym(void* handle, const char* symbol);
int dlclose(void* handle);
```

And link with `libdl`, i.e. `gcc ... -ldl`
C/C++ Function Pointers

```c
void* (*real_malloc)(size_t) = NULL;
```
C/C++ Function Pointers

```c
void* (*real_malloc)(size_t) = NULL;
```

- Function return type

or using a custom type

```c
typedef void* (*malloc_ptr)(size_t);
malloc_ptr real_malloc = NULL;
```
C/C++ Function Pointers

void* (*real_malloc) (size_t) = NULL;

- Function return type
- Variable name
C/C++ Function Pointers

```c
void* (*real_malloc)(size_t) = NULL;
```

- Function return type
- Variable name
- Function parameter types
C/C++ Function Pointers

void* (*real_malloc)(size_t) = NULL;

- Function return type
- Variable name
- Function parameter types
- Initial value
C/C++ Function Pointers

```c
void* (*real_malloc)(size_t) = NULL;

or using a custom type

typedef void* (*malloc_ptr)(size_t);
malloc_ptr real_malloc = NULL;
```

- Function return type
- Variable name
- Function parameter types
- Initial value
# define _GNU_SOURCE
#include <dlfcn.h>

// Inside the wrapper function
{
    static malloc_ptr real_malloc = NULL;
    real_malloc = (malloc_ptr) dlsym(RTLD_NEXT, "malloc");
}
Assignment №3

- In the malloc/free wrappers in mallocWrap.c:
  - Track memory management information
  - Redirect work to the real malloc and free;
- Upon exit, print all pointers (and sizes) that were not free’d;
- You will need to be notified when the programme ends ⇒ atexit()

Hint: Be careful about using malloc/free yourself (indirectly).

Sample solution: https://os.inf.tu-dresden.de/Studium/SysProg/SS2023/debugging/mallocWrap.c
An anecdote

1. Bug report on strange sound on mp3 flash website
An anecdote

1. Bug report on strange sound on mp3 flash website
2. Located in libflashplayer.so
An anecdote

1. Bug report on strange sound on mp3 flash website
2. Located in libflashplayer.so
3. Reason: Use of memcpy for overlapping regions
An anecdote

1. Bug report on strange sound on mp3 flash website
2. Located in libflashplayer.so
3. Reason: Use of `memcpy` for overlapping regions
4. Should use `memmove`, but plugin is closed source
Linus’ Workaround

http://bugzilla.redhat.com/show_bug.cgi?id=638477#c38

1. Write your own memcpy similar to memmove
2. gcc -O2 -c mymemcpy.c
3. ld -G mymemcpy.o -o mymemcpy.so
4. LD_PRELOAD=mymemcpy.so /opt/google/chrome/google-chrome &
Valgrind

Binary recompilation framework (Valgrind core) with various tools:

- **MemCheck**  memory checks (default)
- **CacheGrind**  cache profiling
  - **Callgrind**  call graph analysis
  - **Helgrind**  race condition detection

How do you pronounce “Valgrind”? (from FAQ)
The "Val" as in the word “value”. The "grind" is pronounced with a short "i" — ie. “grinned” (rhymes with “tinned”) rather than “grined” (rhymes with “find”). Don’t feel bad: almost everyone gets it wrong at first.
Valgrind

Binary recompilation framework (Valgrind core) with various tools:

**MemCheck**  memory checks (default)

**CacheGrind**  cache profiling

  **CallGrind**  call graph analysis

  **Helgrind**  race condition detection

---

**How do you pronounce “Valgrind”? (from FAQ)**

The “Val” as in the word “value”. The “grind” is pronounced with a short “i” — ie. “grinned” (rhymes with “tinned”) rather than “grined” (rhymes with “find”). Don’t feel bad: almost everyone gets it wrong at first.
Assignment №4

Analyze some programmes with Valgrind:

- Get https://os.inf.tu-dresden.de/Studium/SysProg/SS2023/debugging/valgrind.tar.xz
- Use build.sh
https://os.inf.tu-dresden.de/Studium/SysProg/SS2023/debugging/compiler.tar.xz

### scan-build

1. Install the Clang static analyser (e.g. `apt install clang-tools-<version>`)  
2. Run `scan-build make` to analyse code  
3. Run `scan-view` to see the report

### Lists of static analysers

- [https://spinroot.com/static/](https://spinroot.com/static/)  
Compiler Sanitisers

Additional libraries which are able to detect race conditions, memory bugs, undefined behavior, ...

Assignment №5: Address Sanitiser

1. Install libasan (e.g. apt install libasan<version>)
2. Run make asan
   (re-builds all programmes with -fsanitize=address)
3. Run the programmes

Details: man gcc and search for -fsanitize
The GNU Debugger

Interactive debugger (gdb):
- Breakpoints, Watchpoints
- Single-stepping, Reverse-stepping
- Inspect/modify registers & memory
- Scripting

Best with binaries containing debug info, e.g. compiled with the `-g` (or, better, `-ggdb3`) option
Basics

- `r[un] [args] [>...]] [<...]]`
- `start [args] [>...]] [<...]]`
- `starti [args] [>...]] [<...]]`
- `q[uit]`
- `h[elp] [command]`
Breakpoints & Watchpoints

- `break` {function | line | *address} [if condition]
- `watch` {variable | *address}
- `info {break | watch}
- `commands {id(s)}
- `continue`
Inspecting the Programme

- `l[ist] [+|−] [N]` — show programme code
- `disassemble` — disassemble current function
- `i[nfo] reg[isters]` — show register content
- `p[rint] [/FMT] {variable | expression}` — evaluate and print variable or expression
- `disp[lay] [/FMT] {variable | expression}` — evaluate and print every time the programme stops
- `x/FMT {address}` — examine memory
- `bt` — backtrace
Going Forward

- `s[tep]` — step to next source line
- `s[tep]i` — step to next assembler instruction
- `n[ext]` — step to next source line, proceeding through function calls
- `n[ext]i` — step to next assembler instruction, proceeding through function calls
- `fin[ish]` — run to return from current function
Going Backwards

- `record full` — start full execution recording
- `record stop` — stop execution recording
- `rs[tep]` — step to previous source line
- `rs[tep]i` — step to previous assembler instruction
- `rn[ext]` — step to previous line, proceeding through function calls
- `rn[ext]i` — step to previous assembler instruction, proceeding through function calls

See also: https://rr-project.org/
Remote Debugging

- GDB can connect to remote GDB servers
  - Via TCP or serial line
  - `set target remote {address:port}`
- Heavily used in OS/embedded development
- Qemu, Bochs/x86, Valgrind, etc. contain their own GDB servers
Alternate UI

- [tui] layout {asm | src | regs}
- https://github.com/cyrus-and/gdb-dashboard
- https://sourceware.org/gdb/wiki/GDB Front Ends
Scripting

• Run `gdb -ex {gdb_command}`
• Write GDB commands into a text file & run `gdb -x {file}`
• `define mycommand`
• Python API
Assignment №6

https://os.inf.tu-dresden.de/Studium/SysProg/SS2023/debugging/gdb.tar.xz

There are 4 versions of the Sieve of Eratosthenes

But only one works properly

What's wrong with the rest?
System call `ptrace()`

- Child allows parent to intercept child interactions by
  \[ \text{ptrace(PTRACE_TRACEME, 0, 0, 0);} \]
- Parent/Debugger inspect and modifies child state by `ptrace` requests:
  - PEEK/POKE
  - SETREGS/GETREGS
  - CONT/SYSCALL/SINGLESTEP
But I Have no Source Code?!

Hmm, there was this GDB command . . .
But I Have no Source Code?!  

Hmm, there was this GDB command ...  

`disassemble` — disassemble current function

```
400d4e: 55  push  %rbp
400d4f: 48 89 e5 mov  %rsp,%rbp
400d52: bf 84 79 48 00 mov  $0x487984,%edi
400d57: e8 54 6b 00 00 callq  4078b0 <_IO_puts>
400d5c: 5d  pop  %rbp
400d5d: c3  retq
```

Also for:

- Checking what your compiler actually produced
- System programming (e.g. kernel entry/exit)
- Direct hardware control (using specific instructions)
But I Have no Source Code?!

Hmm, there was this GDB command ... 

disasemble — disassemble current function

```
400d4e: 55  push  %rbp
400d4f: 48 89 e5 mov  %rsp,%rbp
400d52: bf 84 79 48 00 mov $0x487984,%edi
400d57: e8 54 6b 00 00 callq 4078b0 <_IO_puts>
400d5c: 5d  pop  %rbp
400d5d: c3  retq
```

Also for:

- Checking what your compiler actually produced
- System programming (e.g. kernel entry/exit)
- Direct hardware control (using specific instructions)
General Purpose Registers

- Data registers
- Flags register
- Instruction pointer

Details: Intel 64 and IA-32 Architectures Software Developer’s Manuals
Register Names

Did you know register names are there for a reason?

- (R/E)SP — stack pointer
- (R/E)BP — base pointer
- (R/E)IP — instruction pointer
Register Names

Did you know register names are there for a reason?

- (R/E)SP — stack pointer
- (R/E)BP — base pointer
- (R/E)IP — instruction pointer
- (R/E)AX — accumulator
- (R/E)BX — base register
- (R/E)CX — counter register
- (R/E)DX — extended accumulator
- (R/E)SI — source index
- (R/E)DI — destination index
Move Instructions

Move data between registers or to/from memory

\[
\begin{align*}
\text{movl} & \quad $1, \%eax \\
\text{movl} & \quad $0xff, \%ebx \\
\text{movl} & \quad (%ebx), \%eax \\
\text{movl} & \quad 3(%ebx), \%eax
\end{align*}
\]
## Assembler Dialects

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>AT&amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>order</strong></td>
<td>instr dest, src</td>
<td>instr src, dest</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>implicit (by register name)</td>
<td>explicit (by instruction)</td>
</tr>
<tr>
<td><strong>Sigils</strong></td>
<td>automatic</td>
<td>prefixes ($, %)</td>
</tr>
<tr>
<td><strong>mem. access</strong></td>
<td>[base + index * scale + disp] [base + disp]</td>
<td>disp(base,index, scale) disp(base)</td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>Intel</th>
<th>AT&amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov eax, 1</code></td>
<td><code>movl $1, %eax</code></td>
</tr>
<tr>
<td><code>mov ebx, 0ffh</code></td>
<td><code>movl $0xff, %ebx</code></td>
</tr>
<tr>
<td><code>mov eax, [ebx]</code></td>
<td><code>movl (%ebx), %eax</code></td>
</tr>
<tr>
<td><code>mov eax, [ebx+3]</code></td>
<td><code>movl 3(%ebx), %eax</code></td>
</tr>
</tbody>
</table>
Arithmetic Operation

Addition / Subtraction

```
add     $1, %eax
add     %eax, %ebx
sub     $1, %eax
sub     %eax, %ebx
```
Arithmetic Operation

Addition / Subtraction

```
add    $1, %eax
add    %eax, %ebx
sub    $1, %eax
sub    %eax, %ebx
```

Where to store the result?
Comparing two Values

cmp $0, %eax

cmp %eax, %ebx
Comparing two Values

\[
\text{cmp } 0, \%eax \\
\text{cmp } \%eax, \%ebx
\]

Where to store the result?
Flags Register

Special purpose register that contains several bits to indicate the result of certain instructions, e.g. `cmp`

- **0** CF — Carry Flag
- **2** PF — Parity Flag
- **6** ZF — Zero Flag
- **7** SF — Sign Flag
- **8** TF — Trap Flag (single step)
- **9** IF — Interrupt Enable Flag

Logical Operation

- `and %eax,%ebx`
- `test %eax,%ebx`
- `or %eax,%ebx`
- `xor %eax,%ebx`
(Conditionally) Jump to an Address

jmp 0xC0FFEE
jmp %eax

Using the flags register...

ja 0xC0FFEE
jae 0xC0FFEE
jb[e] 0xC0FFEE
jg[e] 0xC0FFEE
jl[e] 0xC0FFEE
jne 0xC0FFEE
jz 0xC0FFEE

Details: https://www.felixcloutier.com/x86/jcc
Stack Operations

Push or pop register content to or from the stack

push %eax
pop %eax
pusha
popa
Function-related Operations

Call a function or return from one

call 0xC0FFEE

call 0xBADA55

ret
Describe the high-level function call interface

- How to pass parameters
- Which registers the called function must preserve
- Who does prepare/restoring the stack

Details: https://en.wikipedia.org/wiki/X86_calling_conventions
Function arguments are passed on the stack

Integer values and memory addresses are returned in the EAX register

Registers EAX, ECX, and EDX are caller-saved (volatile)

Other registers are callee-saved (non-volatile)
x86_64 aka AMD64 aka Intel 64 aka x64 (but *not* IA-64)

<table>
<thead>
<tr>
<th></th>
<th>Parameters in Registers</th>
<th>Par. Stack Order</th>
<th>Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td>RCX, RDX, R8, R9</td>
<td>RTL(C)</td>
<td>Caller</td>
</tr>
<tr>
<td>System V</td>
<td>RDI, RSI, RDX, RCX, R8, R9</td>
<td>RTL(C)</td>
<td>Caller</td>
</tr>
</tbody>
</table>
### x86_64 aka AMD64 aka Intel 64 aka x64 (but not IA-64)

<table>
<thead>
<tr>
<th></th>
<th>Parameters in Registers</th>
<th>Par. Stack Order</th>
<th>Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microsoft</strong></td>
<td>RCX, RDX, R8, R9</td>
<td>RTL(C)</td>
<td>Caller</td>
</tr>
<tr>
<td><strong>System V</strong></td>
<td>RDI, RSI, RDX, RCX, R8, R9</td>
<td>RTL(C)</td>
<td>Caller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Return</th>
<th>Callee Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microsoft</strong></td>
<td>RAX</td>
<td>RBX, RBP, RDI, RSI, R12 – R15</td>
</tr>
<tr>
<td><strong>System V</strong></td>
<td>RAX</td>
<td>RBX, RBP, R12 – R15</td>
</tr>
</tbody>
</table>
Interlude: Buffers on the Stack

Stolen from DOS...
The Battlefield: x86/32

General-purpose registers

EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP

Instruction pointer

EIP

Segment, FPU, control, MMX, … registers

CPU

Address Space

Address
Text
Data
BSS
Stack
Kernel

0xFFFFFFFF
0xBFFFFFFF
0x00000000
The Stack

- Stack frame per function
  - Set up by compiler-generated code
- Used to store
  - Function parameters
  - If not in registers – GCC: __attribute__((regparm(<num>)))
  - Local variables
  - Control information
    - Function return address

Address Space

- Text
- Data
- BSS
- Stack
- Kernel

Exploitz
Calling a function

```c
int sum(int a, int b)
{
    return a+b;
}

int main()
{
    return sum(1,3);
}
```

```assembly
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  popl %ebp
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```
%<reg> refers to register content

Offset notation: X(%reg) == memory
Location pointed to by reg + X

Constants prefixed with $ sign

(%<reg>) refers to memory location
pointed to by <reg>

Exploitz
Assembly recap'd

sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  popl %ebp
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
So what happens on a call?

sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret

Exploitz

Stack

EIP

ESP

EBP
So what happens on a call?

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```
So what happens on a call?

sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
So what happens on a call?

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

Stack

EBP (main)

ESP

EBP

EIP

Exploitz
So what happens on a call?

```assembly
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

Stack

EBP (main)

3

ESP

EBP

EIP

Exploitz
So what happens on a call?

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```
So what happens on a call?

Exploitz

```asm
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

Stack

EIP

ESP

EBP (main)

Return Addr

EBP

Exploitz

3

1

Return Addr

EBP

3

1

Return Addr

EBP

Exploitz

3

1

Return Addr

EBP
So what happens on a call?

Exploitz

sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret

Stack

EBP (main)
  EBP
  3
  1
  Return Addr
  EBP (sum)

EIP

ESP

Exploit
So what happens on a call?

Exploitz

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```
So what happens on a call?

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

Stack:
- EBP (main)
  - 3
  - 1
  - Return Addr

- EBP (sum)
- EAX: 3

- EIP

Exploitz
So what happens on a call?

```
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
So what happens on a call?

ExploitZ

Main:
```assembly
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```

Sum:
```assembly
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret
```
So what happens on a call?

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```
Now let's add a buffer

```c
int foo()
{
    char buf[20];
    return 0;
}

int main()
{
    return foo();
}
```

The assembly code for foo:
```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $32, %esp
    movl $0, %eax
    leave
    ret
```

The assembly code for main:
```
main:
    pushl %ebp
    movl %esp, %ebp
    call foo
    popl %ebp
    ret
```
Now let's add a buffer

```assembly
foo:
   pushl %ebp
   movl %esp, %ebp
   subl $32, %esp
   movl $0, %eax
   leave
   ret

main:
   pushl %ebp
   movl %esp, %ebp
   call foo
   popl %ebp
   ret
```

Stack

- EIP
- ESP
- EBP (main)
- EBP(foo)

Return Addr
Now let's add a buffer

foo:
   pushl %ebp
   movl %esp, %ebp
   subl $32, %esp
   movl $0, %eax
   leave
   ret

main:
   pushl %ebp
   movl %esp, %ebp
   call foo
   popl %ebp
   ret

Stack
EIP
ESP
EBP (main)
EBP(foo)
buf
EBP
Return Addr
Exploitz
Calling a libC function

```
int foo(char *str)
{
    char buf[20];
    strcpy(buf, str);
    return 0;
}

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

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret

Exploitz
Calling a libC function

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret

Exploitz

Stack
EBP (main)
string ptr
Return Addr
EBP/foo

EIP
EAX
ESP
EBP
string ptr
EBP(foo)

73 / 90
Calling a libC function

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret

Exploitz

Stack

EBP (main)
string ptr
Return Addr
EBP(foo)

EIP

EBP

ESP

EAX

74 / 90
Calling a libC function

```assembly
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, 4(%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

Stack

- EIP
- ESP
- EBP (main)
- string ptr
- EBP(foo)

Memory Layout:

- EAX: `<string ptr>`

Exploitz
Calling a libC function

```assembly
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, 4(%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```
Calling a libC function

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, 4(%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret

Exploitz

Stack

EIP

EBP

ESP

EBP (main)

string ptr

Return Addr

EBP(foo)

EAX: 
<buf ptr>
Calling a libC function

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, 4(%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
Calling a libC function

```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

string = "Hello world"
Inline Assembly

```
asm [volatile] ( AssemblerTemplate
    : OutputOperands
    [ : InputOperands
      [: Clobbers] ]);

int i = 42;
asm volatile ("add %eax, %eax;"
    : "+a"(i)
    : // no other input, just i
    : // no clobber
);
```

Details: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
Register Constraints and Modifiers

```assembly
asm volatile ("add\%0,\%0;" : "+r"(i) );
```

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>any general purpose register</td>
</tr>
<tr>
<td>a</td>
<td>al, ax, eax, rax</td>
</tr>
<tr>
<td>c</td>
<td>cl, cx, ecx, rcx</td>
</tr>
<tr>
<td>D</td>
<td>edi, rdi</td>
</tr>
<tr>
<td>m</td>
<td>memory operand</td>
</tr>
</tbody>
</table>
```c
int add(int a, int b) {
    asm volatile ("add %1, %0;"
                   : "+r"(a)
                   : "r"(b)
                   );

    return a;
}
```
Compiler Builtins

GCC (and others) come with special *intrinsics* that map to optimised code

Details:
- [https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html](https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html)
Compiler Builtins

GCC (and others) come with special *intrinsics* that map to optimised code

Examples:

- Common libC functions like `__builtin_memcpy`
- `__builtin_expect`
- `__builtin_popcount`
- `__builtin_prefetch`
- `__builtin_unreachable`
- `__builtin_return_address`

Details:

https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html
CPU Time Stamp Counter

64 bit register counting the clocks since system startup

- Pentium*, early Xeon CPUs: increment with every CPU cycle
- Newer Xeons and Core*: increment at a constant rate
- AMD up to K8: per CPU, increment with every CPU cycle

Spot the problem, anyone?
CPU Time Stamp Counter

64 bit register counting the clocks since system startup

- Pentium*, early Xeon CPUs: increment with every CPU cycle
- Newer Xeons and Core*: increment at a constant rate
- AMD up to K8: per CPU, increment with every CPU cycle

Spot the problem, anyone?
Check CPU flags (lscpu) for constant_tsc.
Reading the TSC

Instruction: `rdtsc` stores TSC in EAX (lower 32 bits) and EDX (higher 32 bits)
Reading the TSC

Instruction: `rdtsc` stores TSC in EAX (lower 32 bits) and EDX (higher 32 bits)

```c
unsigned long long rdtsc() {
    unsigned long long hi, lo;

    asm volatile("rdtsc;"
                 "mov %edx, %0\n\t"
                 "mov %eax, %1\n\t"
                 : "=r" (hi), "=r" (lo)
                 );

    return (hi << 32) | lo;
}
```
Reading the TSC

Instruction: `rdtsc` stores TSC in `EAX` (lower 32 bits) and `EDX` (higher 32 bits)

```c
unsigned long long rdtsc() {
    unsigned long long hi, lo;

    asm volatile("rdtsc;"
                 "mov %edx, %0\n\t"  // hi
                 "mov %eax, %1\n\t"  // lo
                 : "=r" (hi), "=r" (lo)
                 );

    return (hi << 32) | lo;
}

Spot the problem, anyone?
Instruction: \texttt{rdtsc} stores TSC in \texttt{EAX} (lower 32 bits) and \texttt{EDX} (higher 32 bits)

\begin{verbatim}
unsigned long long rdtsc() {
    unsigned long long hi, lo;

    asm volatile("rdtsc;"
        "mov %edx, %0\n"
        "mov %eax, %1\n"
        : "=r" (hi), "=r" (lo)
        : "eax", "edx"
        :
    );

    return (hi << 32) | lo;
}
\end{verbatim}
### Before measurement

```c
unsigned long long rdtsc_pre() {
    unsigned long long hi, lo;

    asm volatile("cpuid\n\t" "rdtsc\n\t" "mov edx, 0\n\t" "mov eax, 1\n\t": "=r" (hi), "=r" (lo)
    : "rax", "rbx", "rcx", "rdx");

    return (hi << 32) | lo;
}
```

### After measurement

```c
unsigned long long rdtsc_post() {
    unsigned long long hi, lo;

    asm volatile("rdtscp\n\t" "mov edx, 0\n\t" "mov eax, 1\n\t" "cpuid\n\t": "=r" (hi), "=r" (lo)
    : "rax", "rbx", "rcx", "rdx");

    return (hi << 32) | lo;
}
```

Details: “How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures”, Gabriele Paoloni
Benchmarking Considerations

• RTSC is not for free
Benchmarking Considerations

- RTSC is not for free
- Interruption by other programmes, migration
  - Own OS: measure in kernel and disable IRQs
  - Linux user space: difficult
    - Set CPU affinity
    - Collect 1000s of samples and ignore outliers
Can you take a look at the bug I just opened?

Uh oh.

Is this a normal bug, or one of those horrifying ones that prove your whole project is broken beyond repair and should be burned to the ground?

It's a normal one this time, I promise.

Ok, what's the bug?

The server crashes if a user's password is a resolvable URL.

I'll get the lighter fluid.

https://xkcd.com/1700/, CC-BY-NC 2.5, Randall Munroe
Image Sources

Slide 3

- United States Navy
- Naval Surface Warfare Center, U.S. Naval History and Heritage Command Photograph.

Slide 37

Intel 64 and IA-32 Architectures Software Developer’s Manuals
Image Sources

Slide 6

- Nenad Stojkovic, flickr: nenadstojkovic, CC-BY 2.0
- https://www.pikrepo.com/fgrza/people-digging-soil-using-shovels
- https://www.flickr.com/photos/wwworks/3377221745, CC-BY 2.0
- Billy Brown, flickr, CC-BY 2.0
- https://openstax.org/, CC-BY 4.0
- https://www.homestratosphere.com/parts-of-shovel/