Cyber Security
Understanding the platform

Chris G. Willcocks
Durham University
With the internet, the battlefield is much larger and more complex than in traditional warfare.

- Think hierarchically

https://www.submarinecablemap.com
Computer architectures are simple, but with clever wiring
1. Understanding the computer architecture (getting close to the metal)
2. Understanding the stack
3. Stack and heap memory vulnerabilities
4. Smashing the stack
5. Dangerous system calls
6. Race condition attacks
7. Timing attacks

Today Zooming in on the threat landscape

Coming up
Getting Close to the Metal
Getting Close to the Metal

Shared

L3

Core
Core
Core
Core
Core
Core
Core
Core

L2

Fast, expensive

L1

SLOW! Cheap

Core
Core
Core
Core

Core
Two types of memory: DRAM & SRAM

1 transistor per bit
- “slow”
- cheap

4+ transistors per bit
- fast (~4 clock cycles)
- expensive
- takes up space on die
Computer Architecture

Fast, expensive, close to core(s)

Cheap, slow, far from core(s)

Figure 11: Abstraction of the basic memory hierarchy for a modern CPU and GPU.
Getting Close to the Metal (GPU)

GDDR5 “SLOW”, Cheap
Vector car_image
Vector output
Kernel( int core_index )
{
    float some_value = 0.001 * ...
    TEMP temp
    ATTRIB col0 = fragment.color
    OUTPUT out = result.color
    TEX temp, tex0, texture[0], 2D
    MUL out, col0, temp
    ...
    compile
    0,0,1,1,0,1,0, ...
Getting Close to the Metal (GPU)

L2 Cache

Instruction State + Data State + Clock = Output State

L1 Cache

~25 billion transistors on a GPU die
Getting Close to the Metal (GPU)

Program output:
Data, Instructions & Transforms

- **Core Core Core Core Core:**
  - Shared
  - L3
  - L2
  - L1

- **SLOW! Cheap**

- **0,0,1,1,0,1,1,0,0,1,1,1,...**
  - 0,0,1,1,0,1,1,0,0,1,1,1,...
  - 0,0,1,1,0,1,1,0,0,1,1,1,...
  - 0,0,1,1,0,1,1,0,0,1,1,1,...
  - 0,0,1,1,0,1,1,0,0,1,1,1,...
  - 0,0,1,1,0,1,1,0,0,1,1,1,...
A simple C program

```c
#include <stdio.h>

int w;

int absolute(int x)
{
    if (x<0)
        return -x;
    return x;
}

int abs_mul(int x, int y)
{
    return absolute(x*y);
}

int main()
{
    int a;
    int b;
    a = 5;
    b = -2;
    w = abs_mul(a, b);
    printf("output = %d\n", w);
    return 0;
}
```

1. Compile .c source to object file
2. Execute program
3. Disassemble object to assembly form
4. View assembly
# Compiling and Executing

```c
#include <stdio.h>

int w;

int absolute(int x)
{
    if (x<0)
        return -x;
    return x;
}

int abs_mul(int x, int y)
{
    return absolute(x*y);
}

int main()
{
    int a;
    int b;
    a = 5;
    b = -2;
    w = abs_mul(a,b);
    printf("output = %d\n", w);
    return 0;
}
```

---

### Source code

- `#include <stdio.h>`
- `int w;`
- `int absolute(int x) {
   if (x<0)
       return -x;
   return x;
}

### Machine Instructions

```assembly
0000000000000681 <main>:
681:  55  push rbp
682:  48 89 e5  mov rbp,rsp
685:  48 83 ec 10  sub rsp,0x10
689:  c7 45 f8 05 00 00 00  mov DWORD PTR [rbp-0x8],0x5
690:  c7 45 fc fe ff ff ff  mov DWORD PTR [rbp-0x4],0xffffffff
697:  8b 55 fc  mov edx,DWORD PTR [rbp-0x4]
69a:  8b 45 f8  mov eax,DWORD PTR [rbp-0x8]
69d:  89 d6  mov esi,edx
69f:  89 c7  mov edi,eax
6a1:  e8 bd ff ff ff  call 663 <abs_mul>
6a6:  89 05 88 09 20 00  mov eax,DWORD PTR [rip+0x200988],eax
6ac:  8b 05 82 09 20 00  mov eax,DWORD PTR [rip+0x200982],eax
6b2:  89 c6  mov esi,eax
6b4:  48 8d 3d 99 00 00 00  lea rdi,[rip+0x99]
6bb:  b8 00 00 00 00  mov eax,0x0
6c0:  e8 6b fe ff ff  call 530 <printf@plt>
6c5:  b8 00 00 00 00  mov eax,0x0
6ca:  c9  leave
6cb:  c3  ret
6cc:  0f 1f 40 00  nop DWORD PTR [rax+0x0]
```

### Voltage signals

0,0,1,1,1,0,0,1,1,1, ...
Understanding Application memory

```c
#include <stdio.h>

int w;

int absolute(int x)
{
    if (x<0)
        return -x;
    return x;
}

int abs_mul(int x, int y)
{
    return absolute(x*y);
}

int main()
{
    int a;
    int b;
    a = 5;
    b = -2;
    w = abs_mul(a, b);
    printf("output = %d\n", w);
    return 0;
}
```

Application memory:
- **Heap Memory**
- **Call Stack**
- **Globals**
  - `int w;`
- **Instructions**
  - `push rbp`  
  - `mov rbp, rsp`
Understanding the Stack

The Stack

LIFO data structure

int w;
Understanding the Stack

The Stack

parent func rbp

int main()

fixed size stack frame

int w;
Understanding the Stack

The Stack

```assembly
0000000000000681 <main>:
681:  55            push rbp
682:  48 89 e5      mov rbp, rsp
685:  48 83 ec 10    sub rsp, 0x10
689:  c7 45 f8 05 00 00 00     mov DWORD PTR [rbp-0x8], 0x5
690:  c7 45 fc fe ff ff ff     mov DWORD PTR [rbp-0x4], 0xfffffffff
697:  8b 55 fc            mov edx, DWORD PTR [rbp-0x4]
699:  8b 45 f8            mov eax, DWORD PTR [rbp-0x8]
701:  89 d6            mov esi, edx
703:  89 c7            mov edi, eax
705:  e8 bd ff ff ff call 663 <abs_mul>
70b:  89 05 88 09 20 00 mov DWORD PTR [rip+0x200988], eax
711:  8b 05 82 09 20 00 mov eax, DWORD PTR [rip+0x200982]
717:  89 c6            mov esi, eax
719:  48 8d 3d 99 00 00 00 lea rdi, [rip+0x99]
71f:  e8 6b 00 00 00 00 mov eax, 0x0
725:  e8 6b fe ff ff call 530 <printf@plt>
72b:  8b 00 00 00 00 mov eax, 0x0
72f:  c9            leave
731:  c3            ret
733:  0f 1f 40 00    nop DWORD PTR [rax+0x8]
```

int main()
fixed size stack frame
parent func rbp

int w;
Understanding the Stack

The Stack

parent func ret

b

a

parent func rbp

int w;
Understanding the Stack

```
0000000000000663 <abs_mul>:
663:   55       push rbp
664:   48 89 e5  mov rbp,rsp
667:   48 83 ec 08  sub rsp,0x8
66b:   89 7d fc  mov DWORD PTR [rbp-0x4],edi
66e:   89 75 f8  mov DWORD PTR [rbp-0x8],esi
671:   8b 45 fc  mov eax,DWORD PTR [rbp-0x4]
674:   0f af 45 f8  imul eax,DWORD PTR [rbp-0x8]
678:   89 c7  mov edi,eax
67a:   e8 cb ff ff ff  call 64a <absolute>
67f:   c9  leave
680:   c3  ret
```

The Stack

```
int w;
```
Understanding the Stack

The Stack

```
0000000000000663 <abs_mul>:
663:  55    push  rbp
664:  48 89 e5  mov  rbp,rsp
667:  48 83 ec 08  sub  rsp,0x8
66b:  89 7d fc  mov  DWORD PTR [rbp-0x4],edi
66e:  89 75 f8  mov  DWORD PTR [rbp-0x8],esi
671:  8b 45 fc  mov  eax,DWORD PTR [rbp-0x4]
674:  0f af 45 f8  imul  eax,DWORD PTR [rbp-0x8]
678:  89 c7  mov  edi,eax
67a:  e8 cb ff ff ff  call  64a <absolute>
67f:  c9  leave
680:  c3  ret
```

int w;
Understanding the Stack

The Stack

parent func ret

y
x
parent func rbp
parent func ret

b
a
parent func rbp

int w;

0000000000000663 <abs.mul>:
663:  55    push rbp
664:  48 89 e5 mov rbp,rsp
667:  48 83 ec 08 sub rsp,0x8
66b:  89 7d fc mov DWORD PTR [rbp-0x4],edi
66e:  89 75 f8 mov DWORD PTR [rbp-0x8],esi
671:  8b 45 fc mov eax,DWORD PTR [rbp-0x4]
674:  0f af 45 f8 imul eax,DWORD PTR [rbp-0x8]
678:  89 c7 mov edi,eax
67a:  e8 cb ff ff ff call 64a <absolute>
67f:  c9 leave
680:  c3 ret
### Understanding the Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000064a &lt;absolute&gt;:</td>
<td></td>
<td>absolute stack frame</td>
</tr>
<tr>
<td>64a:</td>
<td>push rbp</td>
<td></td>
</tr>
<tr>
<td>64b:</td>
<td>mov rbp,rsp</td>
<td></td>
</tr>
<tr>
<td>64e:</td>
<td>mov DWORD PTR [rbp-0x4],edi</td>
<td></td>
</tr>
<tr>
<td>651:</td>
<td>cmp DWORD PTR [rbp-0x4],0x0</td>
<td></td>
</tr>
<tr>
<td>655:</td>
<td>jns 65e &lt;absolute+0x14&gt;</td>
<td></td>
</tr>
<tr>
<td>657:</td>
<td>mov eax,DWORD PTR [rbp-0x4]</td>
<td></td>
</tr>
<tr>
<td>65a:</td>
<td>neg eax</td>
<td></td>
</tr>
<tr>
<td>65c:</td>
<td>jmp 661 &lt;absolute+0x17&gt;</td>
<td></td>
</tr>
<tr>
<td>65e:</td>
<td>mov eax,DWORD PTR [rbp-0x4]</td>
<td></td>
</tr>
<tr>
<td>661:</td>
<td>pop rbp</td>
<td></td>
</tr>
<tr>
<td>662:</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

#### The Stack

- **rbp**
- **rsp**

```
int w;
```
Understanding the Stack

```
000000000000064a <absolute>:
  64a: 55 push rbp
  64b: 48 89 e5 mov rbp,rsp
  64e: 89 7d fc mov DWORD PTR [rbp-0x4],edi
  651: 83 7d fc 00 cmp DWORD PTR [rbp-0x4],0x0
  655: 79 07 jns 65e <absolute+0x14>
  657: 8b 45 fc mov eax, DWORD PTR [rbp-0x4]
  65a: f7 d8 neg eax
  65c: eb 03 jmp 661 <absolute+0x17>
  65e: 8b 45 fc mov eax, DWORD PTR [rbp-0x4]
  661: 5d pop rbp
  662: c3 ret
```

The Stack

```
parent func rbp
parent func ret
```

```
x
parent func rbp
parent func ret
```

```
y
x
parent func rbp
parent func ret
```

```
b
a
parent func rbp
```

```
int w;
```
Understanding the Stack

The Stack

```
int w;
```

```
000000000000064a <absolute>:
64a:  55  push   rbp
64b:  48 89 e5 mov   rbp,rsp
64e:  89 7d fc mov   DWORD PTR [rbp-0x4],edi
651:  83 7d fc 00 cmp   DWORD PTR [rbp-0x4],0x0
655:  79 07  jns   65e <absolute+0x14>
657:  8b 45 fc mov   eax,DWORD PTR [rbp-0x4]
65a:  f7 d8  neg   eax
65c:  eb 03  jmp   661 <absolute+0x17>
65e:  8b 45 fc mov   eax,DWORD PTR [rbp-0x4]
661:  5d  pop   rbp
662:  c3  ret
```
Understanding the Stack

The Stack

parent func rbp
parent func ret

y
x

rbp

b

a

parent func rbp

return address here

int w;

0000000000000663 <abs_mul>:
663: 55 push rbp
664: 48 89 e5 mov rbp, rsp
667: 48 8d a4 24 d8 ef ff lea rsp, [rsp-0x1028]
66e: ff
66f: 48 83 0c 24 00 or QWORD PTR [rsp], 0x0
674: 48 8d a4 24 20 10 00 lea rsp, [rsp+0x1020]
67b: 00
67c: 89 7d fc mov DWORD PTR [rbp-0x4], edi
67f: 89 75 f8 mov DWORD PTR [rbp-0x8], esi
682: 8b 45 fc mov eax, DWORD PTR [rbp-0x4]
685: 0f af 45 f8 imul eax, DWORD PTR [rbp-0x8]
689: 89 c7 mov edi, eax
68b: e8 ba ff ff ff call 64a <absolute>
690: c9 leave
691: c3 ret
Understanding the Stack

The Stack

```assembly
0000000000000663 <abs_mul>:
663:  55           push rbp
664:  48 89 e5     mov rbp,rsp
667:  48 8d a4 24 d8 ef ff lea rsp,[rsp-0x1028]
66e:  ff
66f:  48 83 0c 24 00 or QWORD PTR [rsp],0x0
674:  48 8d a4 24 20 10 00 lea rsp,[rsp+0x1020]
67b:  00
67c:  89 7d fc     mov DWORD PTR [rbp-0x4],edi
67f:  89 75 f8     mov DWORD PTR [rbp-0x8],esi
682:  8b 45 fc     mov eax,DWORD PTR [rbp-0x4]
685:  0f af 45 f8   imul eax,DWORD PTR [rbp-0x8]
689:  89 c7        mov edi,eax
68b:  e8 ba ff ff ff call 64a <absolute>
690:  c9           leave
691:  c3           ret
```

int w;
Understanding the Stack

```
0000000000000681 <main>:
681: 55            push rbp
682: 48 89 e5     mov rbp, rsp
685: 48 83 ec 10  sub rsp, 0x10
689: c7 45 f8 05 00 00 00 mov DWORD PTR [rbp-0x8], 0x5
690: c7 45 fc fe ff ff ff mov DWORD PTR [rbp-0x4], 0xffffffff
697: 8b 55 fc     mov edx, DWORD PTR [rbp-0x4]
69a: 8b 45 f8     mov eax, DWORD PTR [rbp-0x8]
69d: 89 d6         mov esi, edx
6a1: e8 bd ff ff ff call 663 <abs_mul>
6a6: 89 05 88 09 20 00 mov DWORD PTR [rip+0x200988], eax
6ac: 89 c6         mov esi, eax
6b4: 48 8d 3d 99 00 00 00 lea rdi, [rip+0x99]
6bb: b8 00 00 00 00 mov eax, 0x0
6c0: e8 6b fe ff ff call 530 <printf@plt>
6c5: b8 00 00 00 00 mov eax, 0x0
6ca: c9           leave
6cb: c3           ret
6cc: 0f 1f 40 00 00 nop DWORD PTR [rax+0x8]
```

The Stack

```
int w;
```
Understanding the Stack

The Stack

int w;

0000000000000681 <main>:
681: 55           push rbp
682: 48 89 e5    mov rbp, rsp
685: 48 83 ec 10  sub rsp, 0x10
689: c7 45 f8 05 00 00 00  mov DWORD PTR [rbp-0x8], 0x5
690: c7 45 fc fe ff ff ff  mov DWORD PTR [rbp-0x4], 0xfffffffff
697: 8b 55 fc    mov edx, DWORD PTR [rbp-0x4]
6a4: 8b 05 8d 3d 99 00 00 00  lea rdi, [rip+0x99]
6bb: b8 00 00 00 00  mov eax, 0
6c0: e8 6b fe ff ff    call 530 <printf@plt>
6c5: b8 00 00 00 00  mov eax, 0
6ca: c9           leave
6cb: c3           ret
6cc: 0f 1f 40 00  nop DWORD PTR [rax+0x8]
Understanding the Stack

- When program thread starts, the operating system reserves some amount of space for the stack.
  - Stack memory does not grow during runtime.
- Caused by...
  - Badly written recursive functions
  - Too much local memory allocated (esp with multi-threading)
- What happens if we need to allocate a large amount of local memory?
  - We would need to know the size of it at compile time.
  - But what if you wanted it to grow dynamically at runtime without know it beforehand?

```c
int stack_overflow() {
    return stack_overflow();
}
```
Introducing the Heap also called Dynamic Memory

Application memory

Heap Memory

Call Stack

Globals

int w;

Instructions

push rbp

mov rbp, rsp

Heap Memory

Can grow while the application is running

Does not grow
Pointers

- Pointers *x are just integer numbers that point to the memory address.
- We can assign pointers to “the address of” &x variables.
- We can read memory from outside
Using the Heap

- C
  - `malloc`
  - `free`
- C++
  - `new`
  - `delete`
- Java
  - `new`
  - lots of stuff resides on the heap
- Python
  - In CPython, all objects live on the heap

`malloc` manages a pool of memory and *sometimes* makes calls to `sbrk` which zeros out new requests.
Using the Heap

- **C**
  - `malloc`
  - `free`

- **C++**
  - `new`
  - `delete`

- **Java**
  - `new`
  - lots of stuff resides on the heap

- **Python**
  - In CPython, all objects live on the heap

malloc manages a pool of memory and *sometimes* makes calls to `sbrk` which zeros out new requests.

```c
#include <stdlib.h>

int main()
{
    float *ptr;
    ptr = (float*)malloc(3*sizeof(float));
    *ptr = 42;
    free(ptr);
    return 0;
}
```
Using the Heap

- C
  - `malloc`
  - `free`

- C++
  - `new`
  - `delete`

- Java
  - `new`
  - Lots of stuff resides on the heap

- Python
  - In CPython, all objects live on the heap

`malloc` manages a pool of memory and *sometimes* makes calls to `sbrk` which zeros out new requests.
Using the Heap

- Memory not guaranteed to be initialized to zero
- Can malloc memory to same size of some sensitive data

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    float *p;
    int i;

    p = (float*)malloc(100*sizeof(float));
    for (i=0; i<100; ++i)
        p[i] = (float)rand()/(float)RAND_MAX;
    free(p);

    p = (float*)malloc(100*sizeof(float));
    p[0] = 6.283185;
    p[3] = 5.0;

    printf("p[0] = %f\n", p[0]);
    printf("p[1] = %f\n", p[1]);
    printf("p[2] = %f\n", p[2]);
    printf("p[3] = %f\n", p[3]);

    free(p);
    return 0;
}
```

Same size, so likely to get same memory

Where did that come from?
Questions

- Which memory type is faster?
  - Stack or Heap?
- Where does heap memory reside physically?
- Where does stack memory reside physically?
- What happens when stack memory leaves scope?
- What happens when heap memory leaves scope?
  - In Java?
    - `ArrayList<Integer> v = new ArrayList<Integer>();`
    - `v.add(1); v.add(2); v.add(3); ... ;`
  - In C++?
    - `std::vector<int> v;`
    - `v.push_back(1); v.push_back(2); v.push_back(3); ...`
Principles of good software?

... fast, secure, efficient, withstands the test of time, simple, intuitive, ...

  - Minimize attack surface area, fail securely, keep security simple, ...

- [https://github.com/nothings/stb](https://github.com/nothings/stb)
  - Sean Barrett, a legendary software veteran, creator of the famous stb libraries such as stb_image, a tiny single file public domain header that enables you to read JPG, PNG, TGA, BMP, PSD, ... instead of including huge monolithic libraries.
  - “Make it easily usable”.
  - “Do anything you want with it”.

- [https://suckless.org/philosophy](https://suckless.org/philosophy)
  - The suckless group, developers of well-written and extendable software, such as the DWM window manager - just 2149 lines of easy-to-understand C code.
    - “Simplicity, clarity, frugality”
    - “Do one task, do it well”

- [https://www.archlinux.org/](https://www.archlinux.org/) “Keep It Simple, Stupid!”

- Writing secure **software** involves understanding the platform, and all the things that come between you (the developer) and the platform.
The key to writing good, secure, software is to understand the platform.

Hardware is the base platform (for software).

Lots of things get in the way:
- Compilers, Abstractions, Software Libraries, GUIs,
- Operating Systems, User Patches, Web Browsers, ...

“Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities”

Offtopic but relevant, assuming Lemi was a Durham University computer science student, what should Lemi have done?