News From the Field

- Anduril, (fairly) new & huge defense contractor (drones, etc.) …
- Palantir, BigData outfit (e.g., for spy agencies)
- US Army CTO on the new Battlefield Communications Network

“fundamental security” problems & vulns, and should be treated as “very high risk”

"We cannot control who sees what, we cannot see what users are doing, and we cannot verify that the software itself is secure," the memo says.

Source: reuters.com

News From the Field

The memo said the system allows any authorized user to access all applications and data regardless of their clearance level or operational need. As a result, "Any user can potentially access and misuse sensitive" classified information, the memo states, with no logging to track their actions.

News From the Field

The memo said the system allows any authorized user to access all applications and data regardless of their clearance level or operational need. As a result, "Any user can potentially access and misuse sensitive" classified information, the memo states, with no logging to track their actions.

News From the Field

The memo said the system allows any authorized user to access all applications and data regardless of their clearance level or operational need. As a result, "Any user can potentially access and misuse sensitive" classified information, the memo states, with no logging to track their actions.

No logging -> Audit not possible.
No compromise recording -> forensics will be very challenging

Ambient Authority! No separation of privilege. Hence, impossible to approximate the principle of least privilege.

Clear indication that security is an afterthought.

Control Transfers

Control transfers change control flow of programs

- Can be predicated on results of a previous comparison (flag bits)
- Arithmetic, logic instructions also set flags (omitted before for brevity)

Distinction between *jumps* and *calls*

- Jumps simply transfer control with no side effects
- Calls used to implement procedures

Distinction between *direct* and *indirect* transfers (*indirect* also known as computed transfers)

- Direct transfers use relative offsets, indirect transfers are absolute (through a register or memory reference)

Instruction Side Effects

Described in the Intel Programmers Manual

- Human readable text
- Hard to parse/understand for computer programs
- Sometimes incomplete

Use the BAP (Binary Analysis Platform)

- BAP features the BAP intermediate language (BIL)
- Make all side effects explicit
- Supports most x86/x86_64 instructions
- Lacks support for some “obscure” SSE instructions

BIL Example

BAP: Binary An... ×

bap.ece.cmu.edu

Binary Analysis Platform

Home · Meet BAP · Documentation · Support · Download · Links · Credits

binary will do.

```
addr 0x0 @asm "add    %rax,%rbx"
label pc_0x0
T_t1:u64 = R_RBX:u64
T_t2:u64 = R_RAX:u64
R_RBX:u64 = R_RBX:u64 + T_t2:u64
R_CF:bool = R_RBX:u64 < T_t1:u64
R_OF:bool = high:bool((T_t1:u64 ^ ~T_t2:u64) & (T_t1:u64 ^ R_RBX:u64
))
R_AF:bool = 0x10:u64 == (0x10:u64 & (R_RBX:u64 ^ T_t1:u64 ^ T_t2:u64
))
R_PF:bool =
  ~low:bool(let T_acc:u64 := R_RBX:u64 >> 4:u64 ^ R_RBX:u64 in
            let T_acc:u64 := T_acc:u64 >> 2:u64 ^ T_acc:u64 in
            T_acc:u64 >> 1:u64 ^ T_acc:u64)
R_SF:bool = high:bool(R_RBX:u64)
R_ZF:bool = 0:u64 == R_RBX:u64
```

BIL code for `add %rax, %rbx`

Control Transfers (Part I)

Instruction	Condition	Description
`jmp x`	unconditional	Direct or indirect jump
`je/jz x`	ZF	Jump if equal
`jne/jnz x`	$\neg$ ZF	Jump if not equal
`jl x`	SF $\oplus$ OF	Jump if less (signed)
`jle x`	(SF $\oplus$ OF) $\vee$ ZF	Jump if less or equal
`jg x`	$\neg$ (SF $\oplus$ OF) $\wedge \neg$ ZF	Jump if greater (signed)
`jb x`	CF	Jump if below (unsigned)
`ja x`	$\neg$ CF $\wedge \neg$ ZF	Jump if above (unsigned)
`js x`	SF	Jump if negative

Procedures

```
int f(int x) {return   x + 1; }
int g(int x) {return    f(x); }
int h(int x) {return f(x *2); }
```

Procedures (functions) are intrinsically linked to the stack

- Provides space for local variables
- Records where to return to
- Used to pass arguments (sometimes)

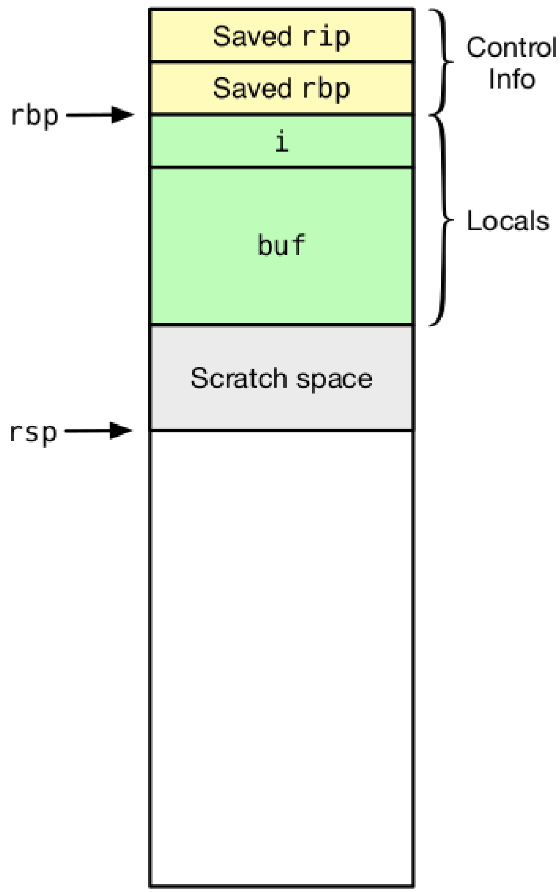
Implemented using stack frames

- Also known as activation records

Control Transfers (Part II)

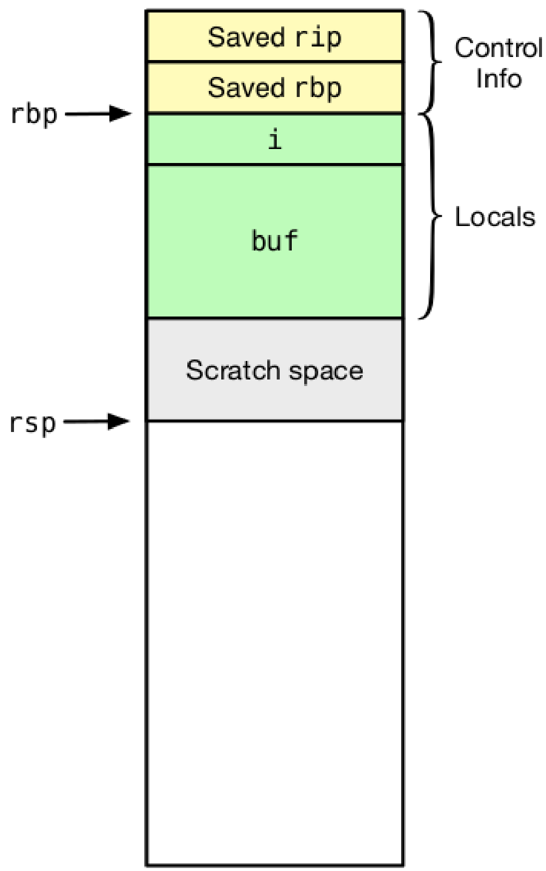
Instruction	Effect	Description
`call x`	rsp ← rsp - 8	Decrement rsp by 8
	Mem(rsp) ← Succ(rip)	Store successor
	rip ← Addr(x)	Jump to address
`ret`	rip ← Mem(rsp)	Pop successor into rip
	rsp ← rsp + 8	Increment rsp by 8

Stack Frame




```c
int auth(const char* user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof(buf));
    buf[sizeof(buf) - 1] = '\0';
    for (i = 0; i < sizeof(buf); i++)
        buf[i] ^= 0xe5;
    return !memcmp(buf, "secret", 6);
}
```

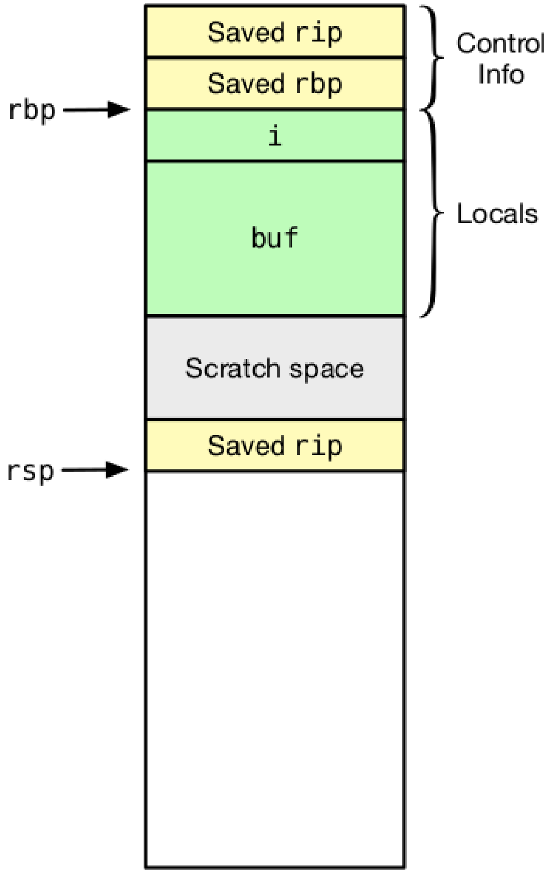
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

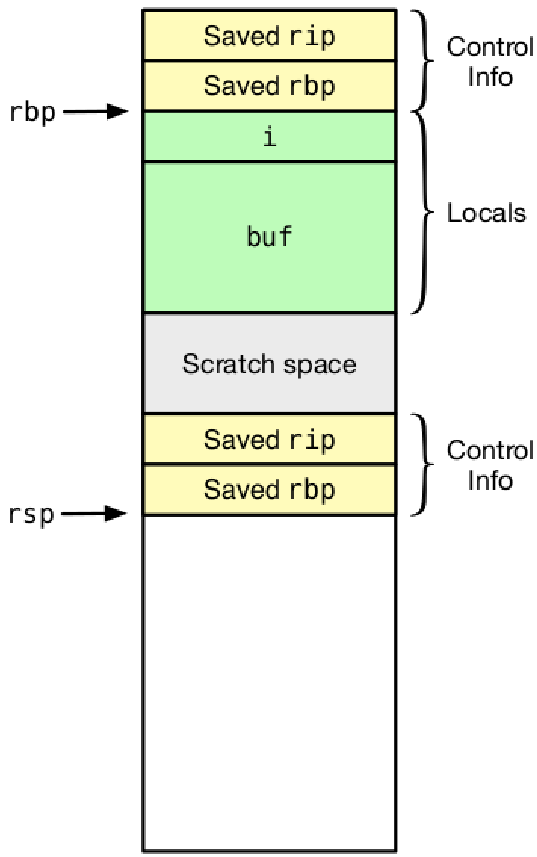
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

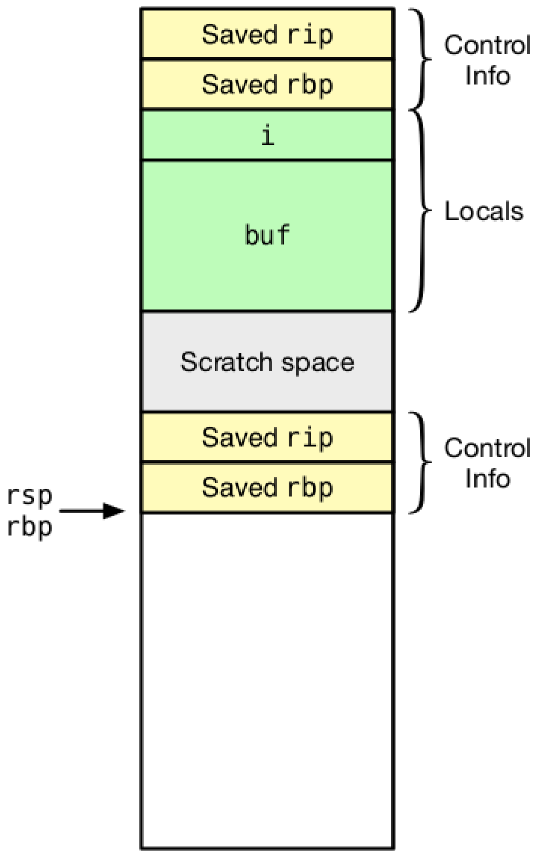
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

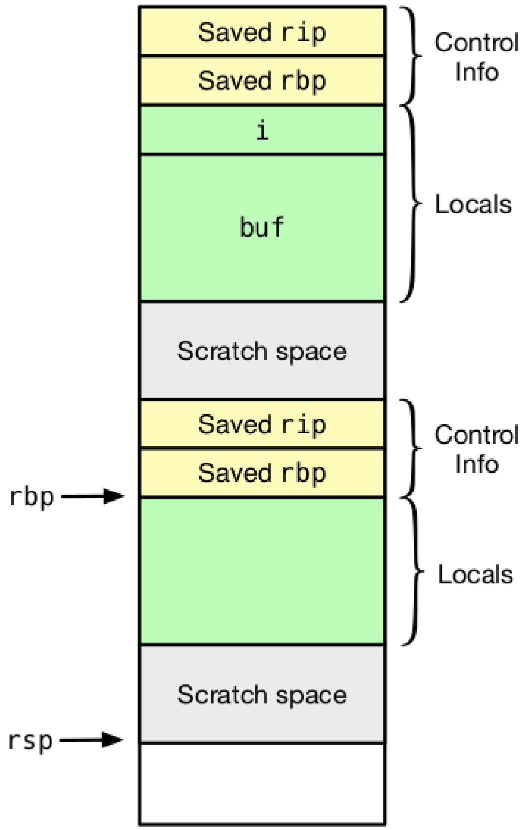
Stack Frame




```
auth:
        ...
        mov rdi, rax
        call strncpy
        mov byte [rbp-0x11], 0x00
        ...

strncpy:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30
        ...
        add rsp, 0x30
        pop rbp
        ret
```

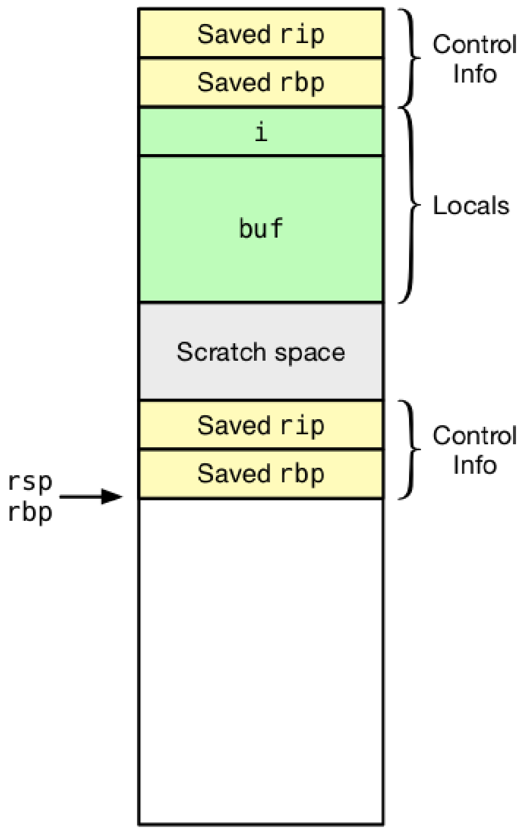
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

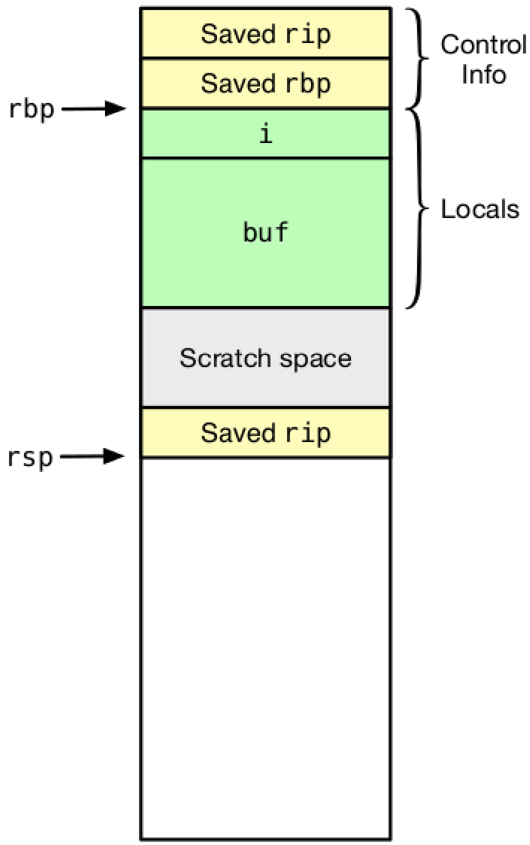
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

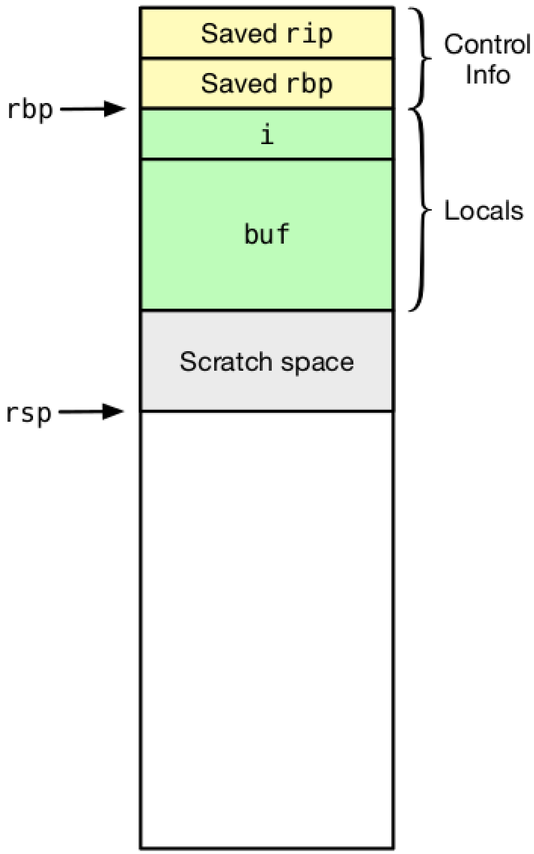
Stack Frame




```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

Stack Frame




```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

Procedure Arguments

Standards (*calling conventions*) exist for argument passing

- Specify where arguments are passed (registers, stack)
- Specify the caller and callee's responsibilities
  - Who deallocates argument space on the stack?
  - Which registers can be clobbered, and who must save them?

Why do we need standards?

- There are many ways to pass arguments
- How would code compiled by different developers and toolchains interoperate?

Calling Conventions

- **We often speak of *callers* and *callees***
  - Caller: Code that invokes a procedure
  - Callee: Procedure invoked by another function
- **Conventions must specify how registers must be dealt with**
  - Could always save them, but that is inefficient (why?)
  - Usually, some registers can be overwritten (clobbered), others cannot
  - Registers that can be clobbered: *caller* saved
  - Registers that must not be clobbered: *callee* saved

cdecl

- **We’ve been concentrating on x86_64, but cdecl is important to know**
  - Linux 32 bit calling convention
- **Arguments**
  - Passed on the stack
  - Pushed *right to left* (reverse order)
- **Registers**
  - eax, edx, ecx are *caller* saved
  - Remainder are *callee* saved
- **Return value in eax**
- **Caller deallocates arguments on stack after return**

stdcall

```
stdcall_fn:
    …
    pop ebp
    ret 0x10 ; return with an operand
```

- Calling convention used by the Win32 API
- Almost identical to cdecl
- **But**, *callee* deallocates arguments on the stack
  - Can you think of a reason why this is better or worse then cdecl? (Hint: `printf()`)

SysV AMD64 ABI

- **x86_64 calling convention used on Linux, Solaris, FreeBSD, Mac OS X**
  - This is what you'll see most often in this course
- **First six arguments passed in registers**
  - rdi, rsi, rdx, rcx, r8, r9
    - Except syscalls, rcx → r10
  - Additional arguments spill to stack
- **Return value in rax**

SysV AMD64 ABI Example

```
int auth( const char * user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof (buf));

auth:
    push rbp                    ; save previous frame pointer
    mov rbp, rsp                ; set new frame pointer
    sub rsp, 0x30               ; allocate space for locals (i, buf)
    movabs rdx, 0x10            ; move sizeof(buf) to rdx
    lea rax, [rbp-0x20]         ; get the address of buf on the stack
    mov qword [rbp-0x08], rdi   ; move user pointer into stack
    mov rsi, qword [rbp-0x08]   ; move user pointer back into rsi
    mov rdi, rax                ; move buf into rdi
    call strncpy                ; call strncpy(rdi, rsi, rdx)
    ...
```

Writing in Assembly

```
> yasm -- version
yasm 1.3.0
```

- Ok, enough review, let's write a program
- In keeping with the slides, we'll use an Intel syntax assembler called yasm
  - nasm is equivalent for this course
  - Feel free to use gas if you can't stand Intel syntax
- Let's write the simplest possible program
  - Immediately exit with status code 0

Hello World in Assembly

```
bits 64             ; we are writing a 64-bit program
section .text       ; we will place this in the .text (code) section

extern _exit        ; we are referencing an external function (exit)
                    ; libc functions are prefixed with ‘_’

global _start       ; declare _start as global symbol
                    ; this preserves a symbol table entry
_start:             ; _start is the default ELF entry point
  mov rdi, 0x00     ; zero out rdi, our first argument
  call _exit        ; call exit(rdi=0)
  int3              ; raise a breakpoint trap if we get here
                    ; (we should never get here)
```

Assembling and Linking

```
> yasm -f elf64 -o exit.o exit.asm
> ld -o exit exit.o -lc
> /lib/ld-2.18.so ./exit
> echo $?
0
```

1. We first assemble the program to an object file exit.o
2. We link an ELF exe against libc
3. We run it using a given runtime loader
   – You might need to specify a different path
   – Or, you might not need to specify it on your system
4. It returns 0!

Disassembly

- **Disassembling is the process of recovering assembly from machine code**
  - Not to be confused with decompilation!
  - Requires knowledge of binary format and ISA
- **Distinction between linear sweep and recursive descent disassembly**
  - Linear sweep begins at an address and continues sequentially until the buffer is exhausted
  - Recursive descent disassembly begins at an address and follows program control flow, discovering all reachable code