Ethical Student Hackers

Intro to Assembly

Sheffield Ethical Student Hackers
Breaking into security.

The Legal Bit

- The skills taught in these sessions allow identification and exploitation of security vulnerabilities in systems. We strive to give you a place to practice legally, and can point you to other places to practice. These skills should not be used on systems where you do not have explicit permission from the owner of the system. It is VERY easy to end up in breach of relevant laws, and we can accept no responsibility for anything you do with the skills learnt here.

- If we have reason to believe that you are utilising these skills against systems where you are not authorised you will be banned from our events, and if necessary the relevant authorities will be alerted.

- Remember, if you have any doubts as to if something is legal or authorised, just don't do it until you are able to confirm you are allowed to.

Code of Conduct

- Before proceeding past this point you must read and agree to our Code of Conduct - this is a requirement from the University for us to operate as a society.
- If you have any doubts or need anything clarified, please ask a member of the committee.
- Breaching the Code of Conduct = immediate ejection and further consequences.
- Code of Conduct can be found at https://shefesh.com/downloads/SESH%20Code%20of%20Conduct.pdf

What is assembly?

- Assembly is a human readable version of machine-code that is as close as you can get to the "bare metal"
- Every processor architecture has its own assembly language – some common ones:
  - x86 (The one we are learning today)
  - ARM (In mobile devices and the new Macbooks)
  - RISC-V (A neat, newish open-source architecture)
- Though different architectures have different instructions and registers, many of the concepts are the same
- If you'd like to see a very basic (and quite fictional) assembly language, check out TIS-100!

Why is it useful to know assembly?

- Low level development
  - At the level of operating systems and bootloaders, this is sometimes the only language available!
  - These layers of the stack can often hide hard-to-find security vulnerabilities!
- Near-direct translation of machine-code
  - Binary programs can be disassembled and reverse-engineered
- An understanding at this level helps understand concepts in other languages
  - Systems-programming languages like C/C++ and Rust have some overlap
  - Higher level languages like Python are much farther from this level though

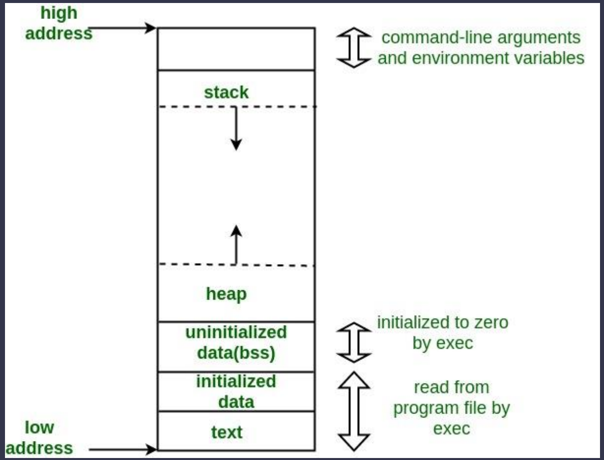
Some basic syntax

```
main:
        mov     $2, %rdi
        cmp     $3, %rdi
        sete    %dil
        movzbq  %dil, %rdi
        test    %rdi, %rdi
        jnz     .T2
        mov     $4, %rbx
        mov     $2, %rcx
        imul    $3, %rcx
        cmp     %rcx, %rbx
        setle   %bl
        movzbq  %bl, %rdi
.T2:
        call    print_bool
        mov     $0, %rax
        ret
print_bool:
        cmp     $0, %rdi
        je      .Lfalse
        mov     $true, %rdi
        jmp     .Lprint
.Lfalse:
        mov     $false, %rdi
.Lprint:
        mov     $0, %rax
        call    printf
        ret
        .data
false:  .string "false\n"
true:   .string "true\n"
```

```
asm_strlen:
	; String passed in rdi.
	; Check for null.
	cmp rdi, 0
	je .zero
	mov rsi, rdi

	.align:
		; Check low 3 bits of current address. If none are set, it's aligned.
		test sil, 7
		jz .done_aligning

		cmp byte[rsi], 0
		je .done
		inc rsi
		jmp .align


	.done_aligning:
		mov r8, 0x7F7F7F7F7F7F7F7F
		mov r9, 0x8080808080808080


	align 8
```

Layout of assembly

- Data section
  - Contains data that is constant once initialised
  - Cannot be changed during execution
- BSS section
  - Used for declaring variables during execution
  - Dynamic, can be changed
- Text section
  - The assembly to execute



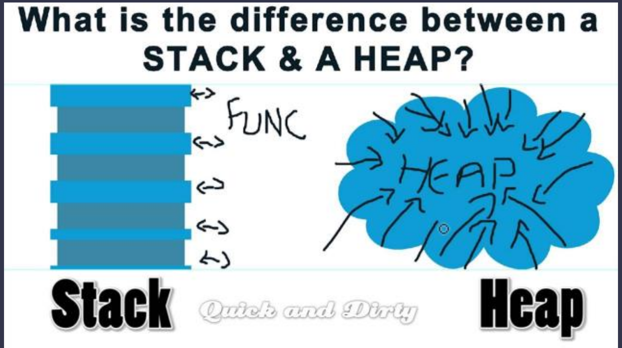
Ethical Student Hackers
SHEFFIELD
Breaking into security.

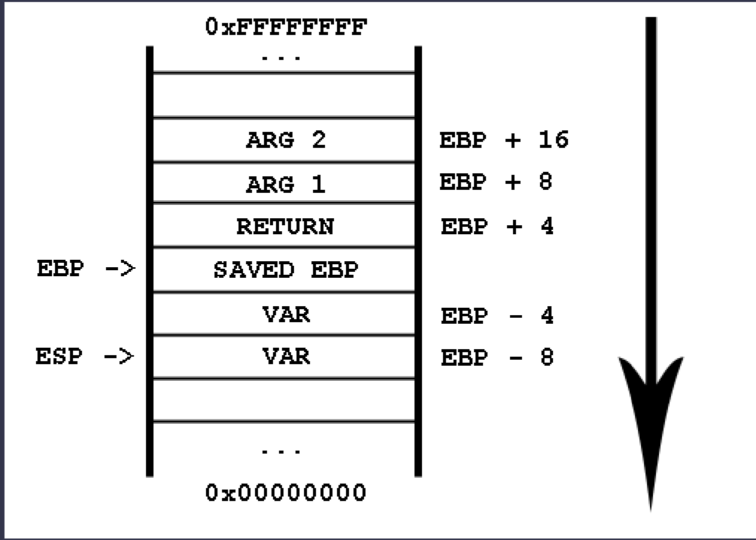
Layout in FASM

```
;"hello, world" in assembly language for Linux
;
;to build an executable:
;       fasm hello.asm

; Output a 64-bit ELF binary
format ELF64 executable

; The .data section of the ELF
segment readable writable

    ; msg is a label, pointing to the first char of our string (with 0xa, a newline, appended)
    msg db 'Hello, world!',0xA

    ; len is a constant label (defined via =) that takes the current address and subtracts the
    ; address of the msg label. This gives the byte-length of the string
    len = $-msg

; The .text section, or code section of the ELF
segment readable executable
; Mark the current address as the executable's entry-point
entry $
    ; Write the string to stdout:
    mov rax,1   ; system call number (sys_write)
    mov rdi,1   ; file descriptor (stdout)
    mov rsi,msg ; message to write
    mov rdx,len ; message length
    syscall     ; call kernel

    ; Exit via the kernel:
    mov rdi,0  ; process's exit code
    mov rax,60 ; system call number (sys_exit)
    syscall    ; call kernel - this interrupt won't return
```

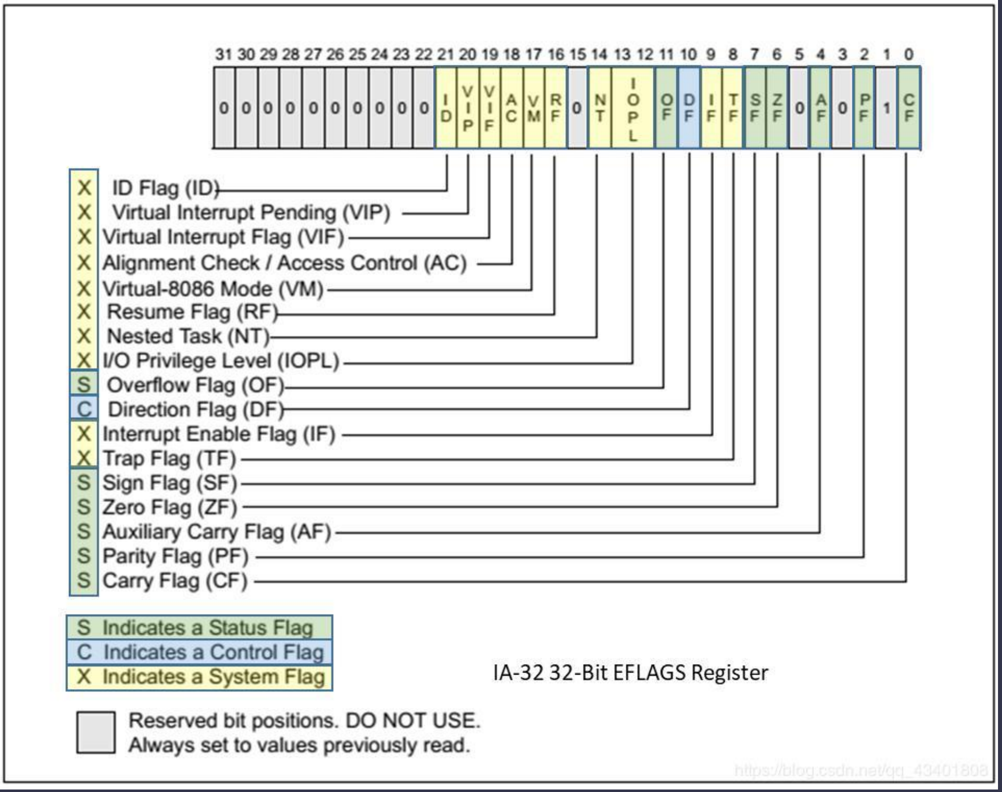
The Stack (Maybe Some Heap Too)

- What is the stack, and heap?
- The stack grows down in memory
- The heap grows up in memory
- Stack frames
  - Growing the stack
  - Restoring the stack





Control Flow



IA-32 32-Bit EFLAGS Register

- `cmp[bwql] src1, src2`
  - Compares src2 to src1 (e.g. `src2 < src1`, `src2 == src1`)
  - Performs (`src2 – src1`) **and sets the condition codes** based on the result
  - src1 & src2 is not changed (subtraction result is only used for condition codes and then discarded)
- `test[bwql] src1, src2`
  - Performs (`src1 & src2`) **and sets condition codes**
  - src2 is not changed
  - Often used with the src1 = src2 (i.e. `test %eax, %eax`) to check if a value is 0 or negative

`jmp label`			
`jmp *(Operand)`			
`je label`	`jz`	`ZF`	Equal / zero
`jne label`	`jnz`	`~ZF`	Not equal / not zero
`js label`		`SF`	Negative
`jns label`		`~SF`	Non-negative
`jg label`	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge label`	`jnl`	`~(SF ^ OF)`	Greater or Equal (signed >=)
`jl label`	`jnge`	`(SF ^ OF)`	Less (signed <)
`jle label`	`jng`	`(SF ^ OF) | ZF`	Less of equal (signed <=)
`ja label`	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae label`	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb label`	`jnae`	`CF`	Below (unsigned <)
`jbe label`	`jna`	`CF | ZF`	Below or equal (unsigned <=)

System Calls

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0	temporary register; used to return long double arguments	No
%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system use (as thread specific data register)	No

```
; Write the string to stdout:
mov rax,1     ; system call number (sys_write)
mov rdi,1     ; file descriptor (stdout)
mov rsi,msg   ; message to write
mov rdx,len   ; message length
syscall       ; call kernel
```

%rax	Name	Entry point	Implementation
1	**write**	sys_write	fs/read_write.c

%rdi	%rsi	%rdx
unsigned int fd	**const char __user** * buf	**size_t** count

https://filippo.io/linux-syscall-table/

Sheffield Ethical Student Hackers
Breaking into security.

Registers

- Registers are a very small location in the CPU that can store and access values very quickly.
  - Very similar to RAM, but a lot faster to access
- They are used to store values while the processor is executing instructions
- Each general-purpose register is 64 bits wide
  - Each 1, 2, 4 and 8 bytes can be accessed individually
- There are other more specialised registers such as the RFLAGS register

Register	Lower byte	Lower word	Lower dword
rax	al	ax	eax
rbx	bl	bx	ebx
rcx	cl	cx	ecx
rdx	dl	dx	edx
rsp	spl	sp	esp
rsi	sil	si	esi
rdi	dil	di	edi
rbp	bpl	bp	ebp
r8	r8b	r8w	r8d
r9	r9b	r9w	r9d
r10	r10b	r10w	r10d
r11	r11b	r11w	r11d
r12	r12b	r12w	r12d
r13	r13b	r13w	r13d
r14	r14b	r14w	r14d
r15	r15b	r15w	r15d

General-Purpose Registers

64-bit	32-bit	16-bit	8 high bits of lower 16 bits	8-bit	Description
RAX	EAX	AX	AH	AL	Accumulator
RBX	EBX	BX	BH	BL	Base index (for use with arrays)
RCX	ECX	CX	CH	CL	Counter for loops and strings
RDX	EDX	DX	DH	DL	Data (commonly extends the A register)
RSI	ESI	SI	N/A	SIL	Source index for string operations
RDI	EDI	DI	N/A	DIL	Destination index for string operations
RSP	ESP	SP	N/A	SPL	Stack Pointer
RBP	EBP	BP	N/A	BPL	Base Pointer (meant for stack frames)
R8.. R15	R8D..R15D	R8W..R15W	N/A	R8B..R15B	General purpose registers 8 to 15

x86 & x86-64	(green)
x86-64 only	(red)

SHEFFIELD Ethical Student Hackers
Breaking into security.

Special Registers

- Instruction pointer register
  - Contains the location of the next instruction

64-bit	32-bit	16-bit	Description
RIP	EIP	IP	Instruction Pointer

- R/E/FLAGS register contains the current state of the CPU
  - Contains useful flags such as Zero, Overflow, Parity, Carry and I/O Privilege level flags
  - https://en.wikipedia.org/wiki/FLAGS_register
- Control registers CR0 to CR7
  - CR0 contains controls for paging, write protections and other things relating to memory
  - CR3 is used for virtual addressing
  - CR4 is used when in protected mode (stops apps writing over each other)

x86 & x86-64	(green)
x86-64 only	(red)

Memory and Addresses

- Memory (or RAM) is a collection of numbered ‘cells’ that are 8 bits in size (1 byte)
  - For example, in the image below you can see the cell 7FF62ECFC128 stores hex 40

- We can access multiple bytes at a time:
  - mov rdi, myNum ; pointer to long
    mov rax, QWORD [rdi+8] ; read *next* long from memory
    ret

    myNum:
        dq 117   ; puts one QWORD in memory   [myNur
        dq 42   ; puts another QWORD in memory   [myNu

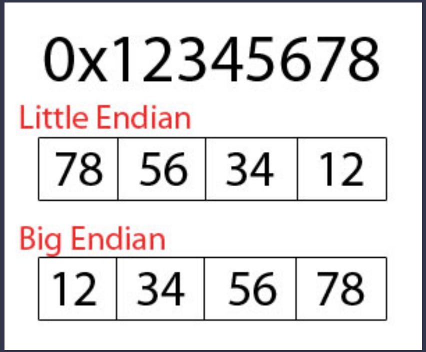
```
Protect:Read/Write  AllocationBase=7FF62ECD0000 Base=7FF62ECFC000 Size=2000 Mo
address       20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 0123456789ABCDEF
7FF62ECFC120 00 00 00 00 00 00 00 00 40 2B CD 2E F6 7F 00 00 ........@+ . ▯..
7FF62ECFC130 A0 1A CD 2E F6 7F 00 00 00 00 00 00 00 00 00 00  . . ▯..........
7FF62ECFC140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC150 00 00 00 00 00 00 00 00 B0 1A CD 2E F6 7F 00 00 ........ . . ▯..
7FF62ECFC160 50 1A CD 2E F6 7F 00 00 10 2D CD 2E F6 7F 00 00 P. . ▯....- . ▯..
7FF62ECFC170 90 2C CD 2E F6 7F 00 00 00 00 00 00 00 00 00 00  , . ▯..........
7FF62ECFC180 A0 57 CD 2E F6 7F 00 00 E0 71 68 9A FB 7F 00 00  W . ▯.. qh  ▯..
7FF62ECFC190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC1A0 40 61 CD 2E F6 7F 00 00 00 00 00 00 00 00 00 00 @a . ▯..........
7FF62ECFC1B0 50 9D 68 9A FB 7F 00 00 F0 58 CD 2E F6 7F 00 00 P h  ▯.. X . ▯..
7FF62ECFC1C0 10 F9 71 9A FB 7F 00 00 00 00 00 00 00 00 00 00 . q  ▯..........
7FF62ECFC1D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC1E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC1F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC210 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 ................
7FF62ECFC220 00 00 68 9A FB 7F 00 00 00 00 00 00 00 00 00 00 ..h  ▯..........
7FF62ECFC230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7FF62ECFC240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

- Windows (and Linux?) have address spaces that are assigned to applications
  - This stops applications overwriting or viewing each others data

Endianness & Bitwise Operations

- Little Endian and Big Endian are simply two ways of representing data

- Operand 1: 0101 1010
  Operand 2: 0011 1001
  - AND Op1, Op2 # Op 1 = 0001 0000
  - OR Op1, Op2 # Op 1 = 0111 1011
  - XOR Op1, Op2 # Op 1 = 0110 0011
  - NOT Op1 # Op 1 = 1010 0101
  - SAR Op1, 3 # Op 1 = 0000 1011 Logical shift Right by 3
  - SAL Op1, 3 # Op 1 = 1101 0000 Logical shift Left by 3



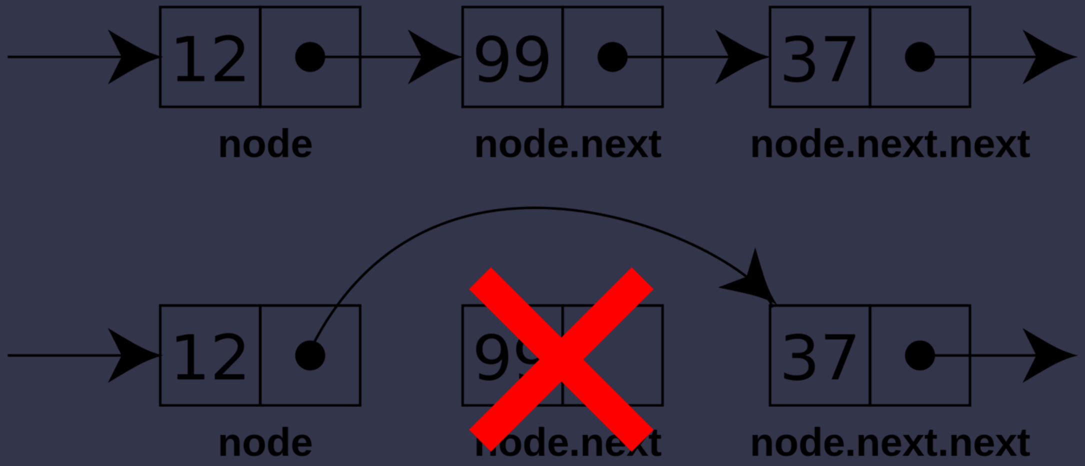
Pointers

- Pointers are a variable that stores the address of another variable

- Really useful for referencing large areas of data
  - We can have a 'base' address, and then reference the data with an offset
  - Last slide shows this, look at 'myNum'

- In assembly, we can reference pointers like so:
  - mov rbx, [rsp]   # Take the value from the address stored in rsp and store it in rbx
    - rsp = 0000021163C3C690
    - 0000021163C3C690 = FFFFFFFFFFFFFFFF
    - So rbx would contain FFFFFFFFFFFFFFFF
  - mov [rsp], rbx   # Take value in rbx and store in the memory address stored in rsp
    - rbx = FFFFFFFFFFFFFFFF
    - rsp = 0000021163C3C690
    - So memory address 21163C3C690 would contain FFFFFFFFFFFFFFFF

More Pointers



Sheffield Ethical Student Hackers

Breaking into security.

Demos Time!

```c
# include <stdio.h>
void main() {
    char operator;
    double a,b;
    printf("Enter an operator (+, -, *,): ");
    scanf("%c", &operator);
    printf("Enter two numbers ");
    scanf("%lf %lf",&a, &b);
    switch(operator)
    {
        case '+':
            printf("%.1lf + %.1lf = %.1lf",a, b, a + b);
            break;
        case '-':
            printf("%.1lf - %.1lf = %.1lf",a, b, a - b);
            break;
        case '*':
            printf("%.1lf * %.1lf = %.1lf",a, b, a * b);
            break;
        case '/':
            printf("%.1lf / %.1lf = %.1lf",a, b, a / b);
            break;
        default:
            printf("Error! operator is not correct");
    }
}
```

Miscellaneous Resources

- https://www.youtube.com/watch?v=DNPjBvZxE3E
- https://sensepost.com/blogstatic/2014/01/SensePost_crash_course_in_x86_assembly-.pdf
- http://www.cs.unc.edu/~porter/courses/cse306/s13/slides/x86-assembly-handout.pdf
- https://blog.adafruit.com/2019/04/10/a-crash-course-in-x86-assembly-for-reverse-engineers-assembly-reverseengineering/
- http://staff.ustc.edu.cn/~bjhua/courses/security/2014/readings/x86.pdf
- https://jakash3.wordpress.com/2010/04/24/x86-assembly-a-crash-course-tutorial-i/
- https://www.cs.tufts.edu/comp/40/docs/x64_cheatsheet.pdf
- https://trailofbits.github.io/ctf/vulnerabilities/references/X86_Win32_Reverse_Engineering_Cheat_Sheet.pdf
- https://bitvijays.github.io/LFC-BinaryExploitation.html
- https://opensource.com/article/20/4/linux-binary-analysis
- https://github.com/slimm609/checksec.sh
- https://cutter.re/
- https://montcs.bloomu.edu/Information/LowLevel/Assembly/hello-asm.html#helloLinux
- https://www.youtube.com/watch?v=NcaiHcBvDR4
- https://filippo.io/linux-syscall-table/

Upcoming Sessions

What's up next?

www.shefesh.com/sessions

1st March: Game Hacking

8th March: Making a CTF

15th March: Web App Hacking

Any Questions?

www.shefesh.com

Thanks for coming!

SHEFFIELD Ethical Student Hackers
Breaking into security.