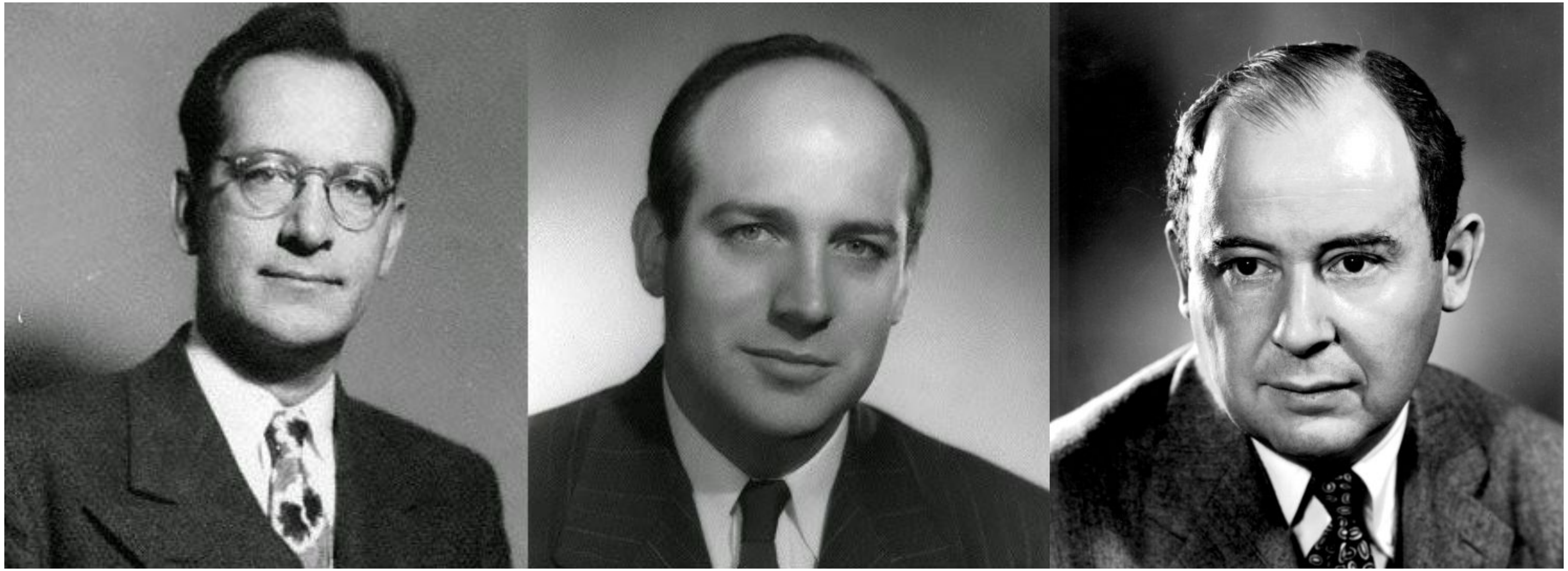
# Module: Shellcoding

Data Execution Prevention

Yan Shoshitaishvili
Arizona State University


John Mauchly (Physicist), John Presper Eckert (Electrical Engineer), John Von Neumann (Mathematician)

`John von Neumann, First Draft of a Report on the EDVAC, 1945.`

CODE

DATA

CODE

DATA

## Shellcode Mitigation: the "No-eXecute" bit

Finally, computer architectures wised up!

Modern architectures support memory permissions:

- **PROT_READ** allows the process to read memory
- **PROT_WRITE** allows the process to write memory
- **PROT_EXEC** allows the process to execute memory

Intuition: *normally*, all code is located in .text segments of the loaded ELF files. There is no need to execute code located on the stack or in the heap.

By default in modern systems, the stack and the heap are *not* executable.

YOUR SHELLCODE NEEDS TO EXECUTE.

Game over?

## A simpler shellcode...

The rise of NX *has* made shellcoding rarer.

It is now... an ancient art!

## Remaining Injection Points - de-protecting memory

Memory can be made executable using the `mprotect()` system call:

1. Trick the program into `mprotect(PROT_EXEC)`ing our shellcode.
2. Jump to the shellcode.

How do we do #1?

- Most common way is *code reuse* through *Return Oriented Programming*. We will cover this in a future module.
- Other cases are situational, depending on what the program is designed to do.

## Remaining Injection Points - JIT

Enter: Just in Time Compilation.

- Just in Time compilers need to generate (and frequently re-generate) code that is executed.
- Pages must be writable for code generation.
- Pages must be executable for execution.
- Pages must be writable for code *re-generation*.

The safe thing to do would be to:

- `mmap(PROT_READ|PROT_WRITE)`
- write the code
- `mprotect(PROT_READ|PROT_EXEC)`
- execute
- `mprotect(PROT_READ|PROT_WRITE)`
- update code
- etc...

## Remaining Injection Points - JIT

System calls are SLOW.

The point of JIT is to be FAST.

SLOW and SAFE tends to lose to FAST.

Writable AND executable pages are common.

If your binary uses a library that has a writable+executable page, that page lives in your memory space!

## Remaining Injection Points - JIT

What if the JIT safely mprotect()s its pages?

Shellcode injection technique: JIT spraying.

- Make constants in the code that will be JITed:
  ```
  var evil = "%90%90%90%90%90";
  ```
- The JIT engine will mprotect(PROT_WRITE), compile the code into memory, then mprotect(PROT_EXEC). Your constant is now present in executable memory.
- Use a vulnerability to redirect execution into the constant.

## Remaining Injection Points - JIT

JIT is used *everywhere*: browsers, Java, and most interpreted language runtimes (luajit, pypy, etc), so this vector is very relevant.

## Mitigation: Sandboxing

What if we accept that the attacker will get code execution, but try to limit the damage they can do?

Stay tuned!