

# Reverse Engineering

## Intro

By [MisterPine](#) with slides based on [IkOri4n's](#)

```
import pwn

pwn.context.arch = "amd64"
pwn.context.os = "linux"

SHELLCODE = pwn.shellcraft.amd64.linux.echo('Test') + pwn.shellcraft
EXPLOIT = 0x45*b"\x90" + pwn.asm(SHELLCODE, arch="amd64", os="linux")

PROGRAM = b""
length = 20 + 16
for i in EXPLOIT:
    PROGRAM += i*b'+ ' + b'>'

    if i == 1:
        length += 5
    elif i > 1:
        length += 6
    length+= 13

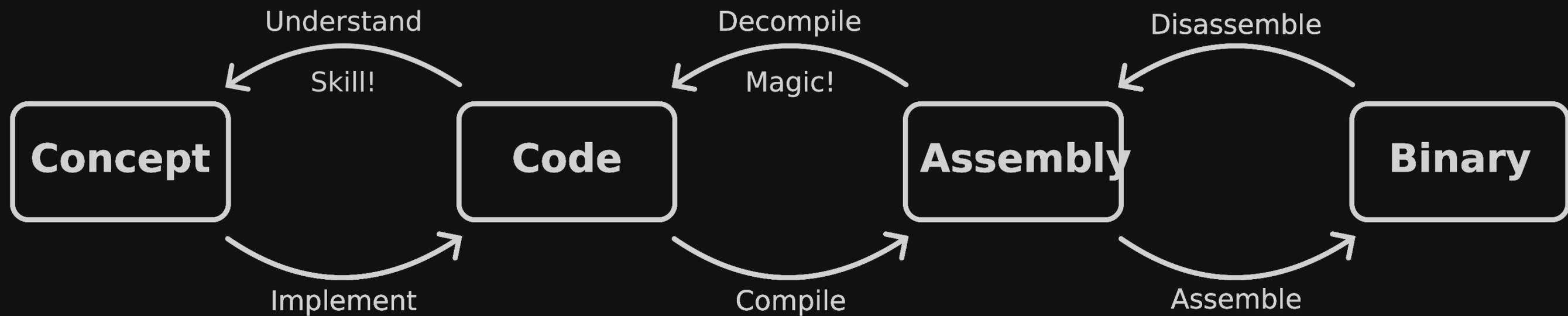
(0x8000 - length) > 0x40:
    PROGRAM += b"<>"
    length += 2*13

    b"."["
    ]

(0x8000 - length) + 7 -1
    (0xF+0x10)*b"<"

(host", 1337) as conn:
    (b"Brainf*ck code: ")
    PROGRAM)
    e()
```

# What are we doing here?



# Why would I need that?

- CTF
- Vulnerability research
- Malware analysis
- No docs, source available
- Modding, Cracking

...plus it's fun!

# Common tools

- **file** type infos based on magic bytes; Good first step for every challenge
- **binwalk** identify & opt. extract embedded files and data
- **strings** dumps strings found in file (sometimes there's a flag in there :P)

# Decompilers

## Open source:

- Ghidra reverse engineering tool created by NSA
- angr management academic binary analysis framework
- cutter reverse engineering tool powered by Rizin

## Commercial:

- Binary Ninja sleek, affordable IDA competitor (free and cloud version)
- IDA pro "gold standard" of disassemblers (expensive)

# Rev player trust issues

Tool output is not always perfect!

- `file` checks *known* magic bytes (first match)
- Decompilers make (wrong) assumptions all the time!
- Tool output may differ (different strengths)

Never trust a tool you haven't written yourself!

Especially don't trust one you have!!!

# Dynamic approach

# Debugging with ~~gdb~~ pwndbg

We don't use plain gdb here!

# Overview

Function	Meaning
<code>help</code>	Print list of commands and specific help
<code>pwndbg</code>	Print list of pwndbg commands
<code>run args</code>	Run the program
<code>starti args</code>	Run the program and break on first instruction
<code>break expr</code>	Break at the given address or symbol
<code>watch expr</code>	Break when a value is written to the given address
<code>rwatch expr</code>	Break when a value is read from the given address
<code>continue</code>	Continue program execution
<code>si</code> and <code>ni</code>	Step into and step over

# Examine Memory

```
x/<amount><format><size> <expr>
```

Parameter	Meaning
<code>amount</code>	Number of things to read
<code>format</code>	Output format, notably x, a, s for hex, addresses, and strings
<code>size</code>	Size of the data blocks, b, h, w, g for 1, 2, 4, 8 bytes respectively
<code>expr</code>	C-like expression describing data location

`telescope [addr] [count]` Recursively dereference pointers (e.g., stack overview)

# Automated debugging

- **gdb Command Files**: run scripts with gdb commands
- **pwntools**: lots of functionality for scripting (see [pwnlib.gdb](#))
- **libdebug**: simple API to debug programmatically

# Dynamic analysis tools

- **strace** trace system calls
- **ltrace** trace library calls
- **gdb** GNU debugger
- **Emulators**

# Other helpful tools

- `angr` symbolic execution
- SMT solvers (e.g., `z3`)

Lots of plugins and tools for specific use cases

# And... Action!

Start playing at [intro.kitctf.de](https://intro.kitctf.de)

Übung macht den Meister!

# Demo alternative



YouTube @stacksmashing

Good quickstart guide & reversing series!