All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them

Native Code Crash Course

WHAT IS NATIVE CODE?

For the purposes of this discussion, native code is any code that is compiled to machine code.

This means most applications you download, most software running on game consoles and other devices, and more.

BREADTH NOT DEPTH

In this session we're going to be touching on a lot of different areas without going deep on any of them.

Resources will be linked on the Hacker101 site if you want to dive deep into any specific area around native hacking.

CODE IS DATA IS CODE

The most important thing to understanding is that there is absolutely no real difference between data and code. You can execute data and you can read/write memory that's normally considered code.
This is fundamental realization that underpins all of native code hacking.

COMPUTER ARCHITECTURE CRASH COURSE

REGISTERS

Registers are small areas of memory located in the CPU itself, where you can store and manipulate data.
In most architectures, many operations can only be executed in the context of registers e.g. reading/writing memory or arithmetic.

There are a small number of registers available for you use, generally from 6-32 depending on the architecture.
This means that much of the code is really going to be moving data from registers to memory and vice versa.

MEMORY

Memory is where most of the state of an application will really be stored.

It can be divided into physical memory (the actual RAM you have available) and virtual memory (an abstraction that allows you to actually access memory).

PHYSICAL MEMORY

Physical memory is usually addressed from 0, so if you have 64MB of RAM, the lowest address will be 0x00000000 and the highest will be 0x03FFFFFF

Applications running on a modern OS will never directly access physical memory, as they would be able to read/write memory for any other program or user.

VIRTUAL MEMORY

Virtual memory is an abstraction through which normal applications will access physical memory.

This is how processes are separated from each other and how we share code between processes.

STACK

Values local to a function get stored in a part of memory called the stack. This is said to grow down, because if you have a stack allocated from 0x10000 to 0x20000, the first value would be written around 0x1fff8 and the last value will be written at 0x10000.

When you call a function, you create a new “stack frame”. You store the previous stack pointer somewhere, then subtract from it give yourself space in which to store any data required for the current function.

Once you’re done executing the function, you restore the original stack pointer.

ENDIANESS

The endianness of a CPU determines the order in which separate bytes of a value are stored in memory. This is needed because memory is accessed in 8-bit units (bytes) and most numbers are 16-bit, 32-bit, 64-bit or even more.

Little endian stores data with the least significant byte first in memory. That is, 0x01ab23cd will end up being stored in memory as four bytes, cd 23 ab 01.

Big endian stores the most significant byte first. The same value on a big-endian system will be stored as: 01 ab 23 cd.

INSTRUCTIONS

This is the basic unit of machine code. An instruction typically performs a single operation. For example:

- Add two registers together
- Compare a register to a number
- Jump to another instruction

PROGRAM COUNTER

The program counter (PC) or instruction pointer (IP) is a register that tells the CPU which instruction to execute next.

On most systems you can’t directly write to this register, but rather use either a jump or call instruction to change it.

CALLS

Calls are a special type of jump which, before changing the program counter, first store the address of the next instruction either to the stack or to a specific register. This is how you call a subroutine (function)

When that subroutine is done, it uses a return instruction to set the program counter back to that stored address.

SYSCALLS

Syscalls are how a normal application tells the kernel to perform an operation e.g. reading/writing a file, sending/receiving network data, or launching other applications.

SHARED LIBRARIES

You could build an application by including every single piece of code you use inside the main binary, which is known as static linking.

However, most applications use dynamic linking, where they call out to other binaries called shared libraries. These are DLLs in Windows, .so files on Linux, or dylibs on Mac.

When the executable gets loaded into memory by the kernel, it loads the referenced shared libraries into the address space as well.

Additionally, applications can load these libraries at runtime; this is often used to load plugins.

DISASSEMBLERS

Disassemblers turn machine code into a readable text form.

Interactive disassemblers are typically GUI-based and allow you to browser through the code and annotate it (e.g. giving the subroutines names).

Non-interactive disassemblers simply give you a text listing of the code.

INTERACTIVE DISASSEMBLERS

- IDA Pro – Gold standard, very expensive
- Hopper – Good for MacOS and iOS apps particularly, much less expensive
- Ghidra – NSA’s open source tool, fairly new. Competes directly with IDA and works well, but still fairly buggy.

NON-INTERACTIVE DISASSEMBLERS

- objdump – Comes with GNU binutils, works well for binaries from Unix-like systems
- ODA – Online disassembler, good for small bits of shellcode
- Radare2 – Fully-featured reverse engineering framework, great disassembler but steep learning curve

DECOMPILERS

Decompilers allow you to turn machine code back into something resembling C. This is often buggy and can hide important details, but it makes reading the code much easier and faster. IDA has a greater decompiler plugin (expensive) and both Hopper and Ghidra include their own by default.

DEBUGGERS

Debuggers allow you to look at a running application and see what the state of its registers, memory, etc are at any point.

- Gdb (GNU Debugger) – Good for Unix-like systems

- WinDbg – Good for Windows
- Most interactive disassemblers also include debuggers

HEX EDITORS

Hex editors allow you to easily manipulate values or machine code inside a binary; some even allow you to view the memory of running applications.

I personally recommend 0xED on MacOS and HxD on Windows, but there are hundreds to choose from.

ASSEMBLERS

Assemblers take assembly code and convert it to machine code. In the context of hacking, this is typically done to procedure small patches for your purposes.

I use the Shell Storm online assembler for this purpose generally.

BUGS

BUFFER OVERFLOWS

Buffer Overflows occur when data is being written to a part of memory without concern for how much is available. For instance, if you write 16 bytes of data into an 8 byte buffer, you’ve written 8 bytes of data over something else.

```
uint8 buffer[4];
uint32 value = 0;
strcpy(buffer, “testAAAA”);
printhex(value); // 0x41414141
```

OUT OF BOUNDS WRITES

This bug occurs when a program writes some value to an attacker-controlled array index without checking that it’s within the bounds of the array.

While an attacker may or may not be able to control the value written, this can be used to corrupt other values in an advantageous way.

```
void vulnerable(int index){

    uint32_t array[16];
    uint32_t value = 0;
    array[index] = 0xDEADBEEF;
    printhex(value); // prints 0xDEADBEEF;
}

vulnerable(16);
```

USE AFTER FREE

This bug occurs when the lifecycle of an object isn't properly handled. If memory is allocated for an object, then the object is freed before the last use, it's possible that a number of different things can occur: information leaks, arbitrary writes, or direct code execution.

PROTECTIONS

STACK CANARIES

Since function return addresses are on the stack, buffer overflows affecting data on the stack can overwrite those return addresses.

Stack canaries prevent this in a really simple way: right before the return address, put a random (known) value. Before the function return, check to see if this value is still right.

NX

On many systems, there's no bit on the page table indicating that it's executable; if it's readable, it's executable. NX fixes this by only allowing pages explicitly marked as code to be executed.

That prevents attackers from simply putting their shellcode somewhere into data memory and jumping to it.

W^X

Sometimes applications need to write new code to memory at runtime, e.g. the browser compiling JavaScript to machine code for performance. When this happens, it's possible for an attacker to simply write their own code there and jump to it.

ASLR

Address Space Layout Randomization is where a process address space will be randomized at creation. The binary and its shared libraries will each get a different random base address, so an exploit can’t just use the same fixed addresses.

On a 32-bit systems, this may only be 10-16 bits of randomness, or 1024-65536 possible base addresses.

On 64-bit systems, though, this can be anywhere from 16-24bits of randomness, or up to 16 million possible base addresses.

BYPASSING PROTECTIONS

JS IS YOUR FRIEND

Given that ASLR is everywhere, being able to adapt your exploit to randomized conditions is key. This is why so many exploits start from a JavaScript engine, e.g. in a browser.

If you have a large attack surface to look at, browsers are always a solid bet. They’re huge, full of bugs, and JS makes it easy.

INFORMATION LEAKS

In order to bypass ASLR, you have two options: guess or know. If you are attacking a service that relaunches immediately after a crash, it may just be easiest to keep throwing the same exploit at it until it triggers; that’s the guess approach.

A much more precise method is to use information leaks to know exactly where things are in memory.

If you can get even a single address to a known point inside a binary, you know where everything important is, and you can tweak your exploit to use that (newly discovered) base address.

HEAP SPRAYING

Heap spraying is a technique you can use when you don’t know exactly where in memory a given exploit payload may need to be located, or if you can’t explicitly put it there.

You allocate many (often thousands) of copies of your exploit object/shellcode and essentially hope that one of them ends up in the right place.

ROP

Return-Oriented Programming is a way to get around not being able to put your shellcode in executable memory. Rather than building shellcode that consists of machine code, you built it up as a call stack, then trigger a return that uses that stack.

Your “code” is then made up of a bunch of gadgets, each consisting of a few instructions followed by a return.

By placing the right values onto the stack, you can perform nearly any operation you can imagine.