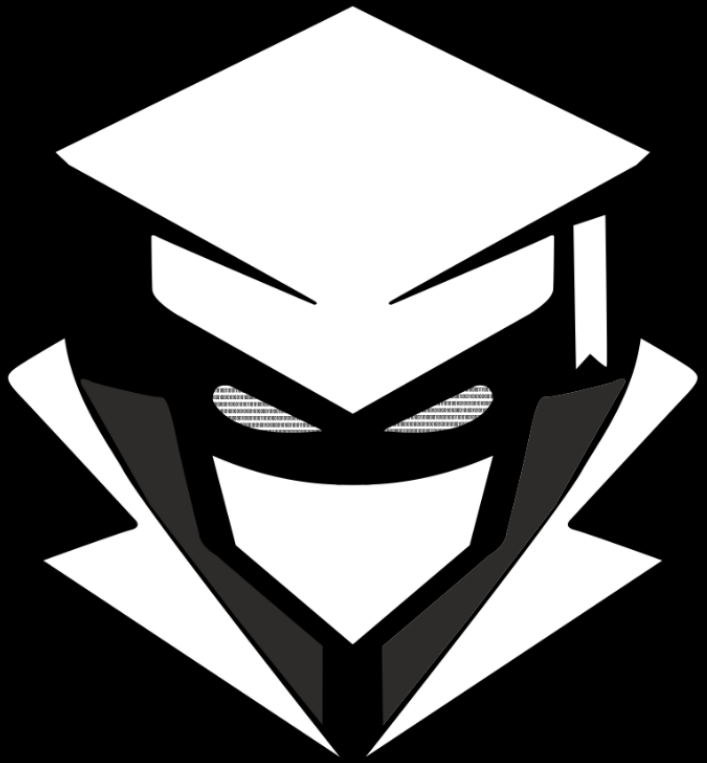

MISC

INTRO TO REVERSE ENGINEERING

accenture

Announcements

- Next Tuesday: Penetration Testing with Pamela O'Shea
  - Time: 12:15pm - 2pm
  - Location: 4.20 KLD (This room)
- Next Thursday: MISC AGM
  - Time: 1:15pm-3pm
  - Location: Arts-West Room 553

What are we doing today?

- Look at how programs go from code, to binaries, to processes.
- Introduce the x86 instruction set and x86_64 architecture
- Analysing Binaries:
  - Static analysis
- Analysing Processes:
  - Dynamic analysis

Whats the big idea?

- For our purposes, reverse engineering relates to performing analysis on a piece of technology when we are not privy to its underlying design.
- We can apply reverse engineering technique to:
  - Hardware
  - Network protocols
  - Software (what we're doing today)

Reverse engineering software

- In compiled languages, a compiler turns human readable code into binary data/bytecode suitable for execution in a given environment.
- javac compiles java into java bytecode
- gcc/g++ compile C/C++ code into binaries to be run on a given instruction set.
- Our interest today is in analysing ELF files, the primary executable format for UNIX operating systems.

The ELF file format: quick summary

- Comprised of ELF Header and a series of sections
- Header: provides information about the format of the program
  - Eg 32 vs 64 bit, endianness, machine type etc.
- Sections:
  - .text: the code (machine instructions)
  - data sections:
    - to store initialised, uninitialised, read only data etc.

What does code look like?

```c
#include <stdio.h>

int main(int argc, char ** argv){

  int i;
  int mynum = 1337;

  for(i=0;i<10;i++){
    printf("%d", mynum);

  }
  return 0;
}
```

What does code look like?

```
push   rbp
mov    rbp,rsp
sub    rsp,0x20
mov    DWORD PTR [rbp-0x14],edi
mov    QWORD PTR [rbp-0x20],rsi
mov    DWORD PTR [rbp-0x8],0x539
mov    DWORD PTR [rbp-0x4],0x0
jmp    0x116e <main+57>
mov    eax,DWORD PTR [rbp-0x8]
mov    esi,eax
lea    rdi,[rip+0xea4]        # 0x2004
mov    eax,0x0
call   0x1030 <printf@plt>
add    DWORD PTR [rbp-0x4],0x1
cmp    DWORD PTR [rbp-0x4],0x9
jle    0x1154 <main+31>
mov    eax,0x0
leave
ret
```

x86 Instruction Set

- Most common architecture used in PCs and servers
- Will be our focus today (Many others exist!)
- Complex instruction set
  - A single instruction can execute several operations (example in a moment)
- Variable length instructions
- Let's take a look at some of the more common instructions

x86_64 Architecture - The stack

- A data structure used to keep track of execution context
- A Last in first out data structure
- 'Grows' towards lower addresses
- Example: Function `foo()` is currently executing. `foo` has its own stack frame in which we might find such things as locally declared variables. We note the base and top of this frame in variables `<base>` and `<top>`.

  From within `foo`, we call `bar()`, which requires 24 bytes to store its local variables. We note `<base>`, then put the value of `<top>` into `<base>`, then subtract 24 from `<top>`

x86_64 Architecture - Registers

- What is a register?
- There are typically 16 64bit registers
- Some have specific usage and purpose, eg:
  - `rip` stores the instruction pointer (the address of the instruction currently being executed)
  - `rsp` stores the address of the top of the stack
- Others are general purpose and used to store any data.

x86 Instruction Set - Instructions

- `mov <destinatin> <source>`
  - Move data into/between registers.
- `push`, `pop` : push/pop values onto/from the stack
- `add`, `sub`, `imul`, `or`, `xor`, `and` : Artithmetic operations
- `jmp`: Move `rip` to a particular instruction
- Thousands more. The Intel x86 Software Developer manuals span 4 (very long) volumes.
- In general, if you run into something you haven't seen before, google it, then take the time to understand exactly what its doing.

Code example revisted - C

```c
#include <stdio.h>

int main(int argc, char ** argv){

  int i;
  int mynum = 1337;

  for(i=0;i<10;i++){
    printf("%d", mynum);

  }
  return 0;
}
```

Code example revisted - ASM

```
push   rbp
mov    rbp,rsp
sub    rsp,0x20
mov    DWORD PTR [rbp-0x14],edi
mov    QWORD PTR [rbp-0x20],rsi
mov    DWORD PTR [rbp-0x8],0x539
mov    DWORD PTR [rbp-0x4],0x0
jmp    0x116e <main+57>
mov    eax,DWORD PTR [rbp-0x8]
mov    esi,eax
lea    rdi,[rip+0xea4]        # 0x2004
mov    eax,0x0
call   0x1030 <printf@plt>
add    DWORD PTR [rbp-0x4],0x1
cmp    DWORD PTR [rbp-0x4],0x9
jle    0x1154 <main+31>
mov    eax,0x0
leave
ret
```

Static Analysis - Tools

- Disassembly : Binary to Asembly
- Decompilation : Binary to Code!
- GDB :
  - Debugger with disassembly capabilities
  - Mostly used in dynamic Analysis
- r2 :
  - Powerfull command line RE platform
  - Steep learning curve
- Ghidra
  - NSA's RE platform recently made public
- IDA
  - Free version for Disassembly
  - Paid version for decompilation
  - Industry standard

Dynamic Analysis Analysis

- It is clear that it may also be useful to observe a program while it is executing.
- Get a clearer idea of its behaviour
- No point understanding every bit of a mechanism if all we need is its output
    - We can extend this principal to the 'black box' approach - compare range of inputs to corresponding outputs to infer behaviour/functionality

Dynamic Analysis – Tools

- GDB
- r2
- IDApro can be attached to gdb
- Windows: OllyDbg
- many more
- Our focus will be on GDB - able to perform basic static and dynamic analysis
- Live demo of previous example

What didn't we cover?

- Other architectures/instruction sets ARM is also vey common as it is widely used in mobile phones
- Obfuscation : creating binaries/code that is deliberately written to be difficult to read/dipher functionality.
- Achieved in a arge number of ways, eg:
  - Using cyrptographic principles within execution
  - Manipulating control flow in delierbately confusing ways (Jump here, do nothing, jump back)
  - Inserting many instructions that do nothing
  - Referencing memory in ways a compiler would never do (fool decompilers)
  - Many More