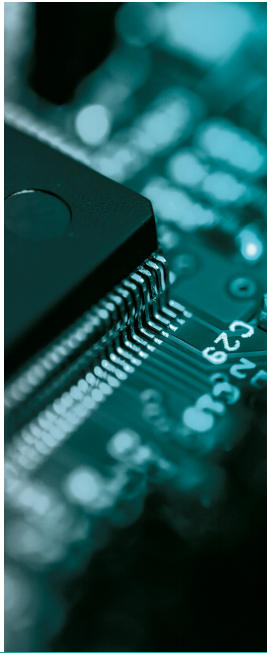
emproof
embedded security

Workshop: Software Reverse Engineering

Tim Blazytko



About Tim

- Chief Scientist, Head of Engineering & Co-Founder of Emproof
- focused on advancing embedded security solutions
- PhD in binary program analysis & reverse engineering
- training and lectures at industry conferences & universities

Today

Reverse Engineering

Techniques

Hands-On

Reverse Engineering

Reverse Engineering

Analysis of binary files (executables, firmware, ...):

- identify functionality
- detect malicious behavior
- reconstruct high-level code
- find bugs
- detect hardcoded keys or credentials
- unlock premium features (cracking)

Compiler

Compiler

- translates high-level code to machine code
- optimizes code for speed and/or performance
- removes information during compilation (comments, symbols, ...)
- generates executable binaries

Binary

Binary

- executable file with code, data, and metadata
- contains CPU-specific machine code
- common formats: ELF (Linux), PE (Windows)

Machine Code

Machine Code

- byte sequences that CPU interprets as instructions
- defined by the CPU's **instruction set architecture** (ISA)
- translated to human-readable form in assembly code

Assembly Code

Assembly Code

- human-readable representation of machine code
- **instruction**: mnemonics and operands
  - **mnemonics**: add, mov, sub
  - **operands**: register or constants
  - **example**: `add r1, r2`
- platform and CPU specific (x86, arm, etc.)

Machine Code and Assembly Code

```
0a 01 0a 00 0b 02 de ad
```

Machine Code and Assembly Code


```
0a 01 0a 00 0b 02 de ad
```

Machine Code and Assembly Code


```
0a 01 0a 00 0b 02 de ad
```

```
add
mul
```

Machine Code and Assembly Code


```
0a 01 0a 00 0b 02 de ad
```

```
add R1
mul R2
```

Machine Code and Assembly Code


```
0a 01 0a 00 0b 02 de ad
```

```
add R1, 0x0a00
mul R2, 0xdead
```

Machine Code and Assembly Code


```
0a 01 0a 00 0b 02 de ad
```

```
add R1, 0x0a00
mul R2, 0xdead
```

The decoded machine code is called assembly code.

Static Analysis

Static Analysis

- examines binary **without executing** it
- reveals structure and functionality
- **common tools**: disassembler & decompiler

Disassembler

Disassembler

- converts machine code into assembly code
- useful for understanding binary's behavior
- key tool in reverse engineering workflows

Disassembler: Decodes Machine Code

```
55 48 89 e5 89
7d fc 89 75 f8
8b 55 fc 8b 45
f8 01 d0 c1 e0
02 5d c3 00 00
```

Disassembler: Decodes Machine Code

```
55 48 89 e5 89
7d fc 89 75 f8
8b 55 fc 8b 45
f8 01 d0 c1 e0
02 5d c3 00 00
```



```
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
shl     eax, 2
pop     rbp
retn
```

Disassembler: Decodes Machine Code

```
55 48 89 e5 89
7d fc 89 75 f8
8b 55 fc 8b 45
f8
02
```

critical step in reverse engineering

```
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
pop     rbp
retn
```

Decompiler

Decompiler

- converts machine code back to high-level code
- aids in understanding program logic and structure
- produces **approximate source code** from binaries

Decompiler: Reconstructs High-Level Code

```
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
shl     eax, 2
pop     rbp
retn
```

Decompiler: Reconstructs High-Level Code

```
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
shl     eax, 2
pop     rbp
retn
```



```
ulong calculate(int param_1,int param_2)
{
    return (ulong)(uint)((param_2 + param_1) * 4);
}
```

Decompiler: Reconstructs High-Level Code

```
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add
shl
pop
retn
```

```
ulong calculate(int param_1,int param_2)
{
    return (ulong)(uint)((param_2 + param_1) * 4);
}
```

eases reverse engineering significantly

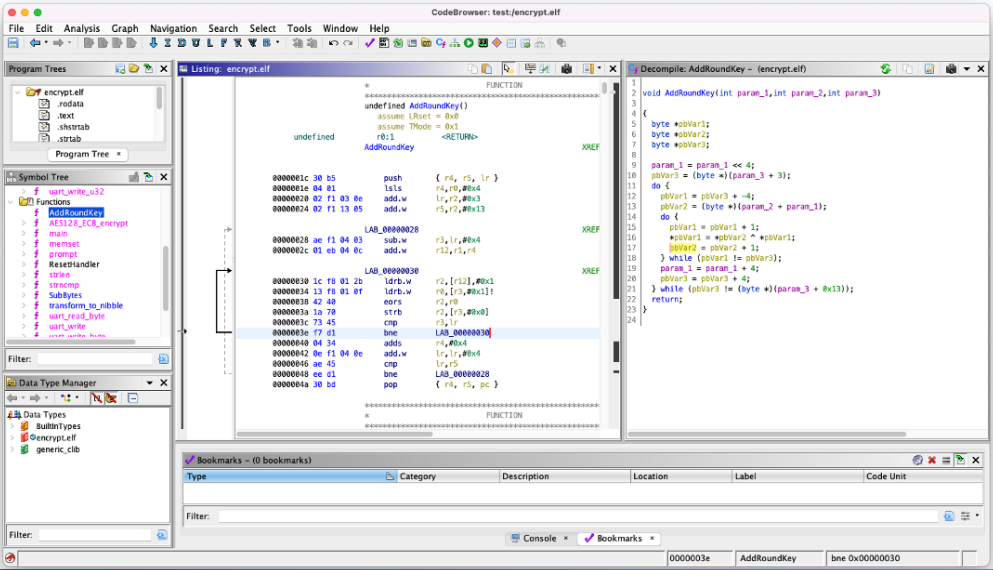
Tools

Ghidra

- open source reverse engineering framework
- powerful disassembler and decompiler
- runs on all common platforms, free to use

`https://ghidra-sre.org`

Ghidra


CodeBrowser: test:/encrypt.elf
File Edit Analysis Graph Navigation Search Select Tools Window Help
Program Trees
encrypt.elf
.rodata
.text
.shstrtab
.strtab
Program Tree
Symbol Tree
Functions
AddRoundKey
AES128_ECB_encrypt
main
memset
prompt
ResetHandler
strlen
strncmp
SubBytes
transform_to_nibble
uart_read_byte
uart_write
Data Type Manager
Data Types
BuiltInTypes
encrypt.elf
generic_clib
Listing: encrypt.elf
undefined AddRoundKey()
assume LRset = 0x0
assume TMode = 0x1
undefined r0:1 <RETURN>
AddRoundKey XREF
0000001c 30 b5 push { r4, r5, lr }
0000001e 04 01 lsls r4,r0,#0x4
00000020 02 f1 03 0e add.w lr,r2,#0x3
00000024 02 f1 13 05 add.w r5,r2,#0x13
LAB_00000028 XREF
00000028 ae f1 04 03 sub.w r3,lr,#0x4
0000002c 01 eb 04 0c add.w r12,r1,r4
LAB_00000030 XREF
00000030 1c f8 01 2b ldrb.w r2,[r12],#0x1
00000034 13 f8 01 0f ldrb.w r0,[r3,#0x1]!
00000038 42 40 eors r2,r0
0000003a 1a 70 strb r2,[r3,#0x0]
0000003c 73 45 cmp r3,lr
0000003e f7 d1 bne LAB_00000030
00000040 04 34 adds r4,#0x4
00000042 0e f1 04 0e add.w lr,lr,#0x4
00000046 ae 45 cmp lr,r5
00000048 ee d1 bne LAB_00000028
0000004a 30 bd pop { r4, r5, pc }
Decompile: AddRoundKey - (encrypt.elf)
void AddRoundKey(int param_1,int param_2,int param_3)
{
byte *pbVar1;
byte *pbVar2;
byte *pbVar3;
param_1 = param_1 << 4;
pbVar3 = (byte *)(param_3 + 3);
do {
pbVar1 = pbVar3 + -4;
pbVar2 = (byte *)(param_2 + param_1);
do {
pbVar1 = pbVar1 + 1;
*pbVar1 = *pbVar2 ^ *pbVar1;
pbVar2 = pbVar2 + 1;
} while (pbVar1 != pbVar3);
param_1 = param_1 + 4;
pbVar3 = pbVar3 + 4;
} while (pbVar3 != (byte *)(param_3 + 0x13));
return;
}
Bookmarks - (0 bookmarks)
Type Category Description Location Label Code Unit
Filter:
Console
Bookmarks
0000003e AddRoundKey bne 0x00000030

Binary Ninja

- commercial reverse engineering framework
- powerful UI, good for patching/cracking
- free version available

`https://binary.ninja/free/`

Tasks

Slides, Setup & Samples

Checkout on GitHub:

https://github.com/emproof-com/workshop_software_reverse_engineering_escar24

Task 1: Hello World

Open `hello_world` in Ghidra.

- Load and analyze the binary.
- Get used to the analysis window (functions, assembly, decompiler).
- Inspect the strings in the binary.
- Locate the main function. What does it do?
- Repeat the task in Binary Ninja.

Task 2: Game Reverse Engineering

- Execute `game` and check how it works.
- Open it in Ghidra and inspect the strings.
- Locate the main function and inspect it in the decompiler.
- Locate the hardcoded license to unlock the full version.

Task 3: Game Cracking

- Open `game` in Binary Ninja and open the main function.
- Identify the variable that hardcodes the number of trials.
- Patch the binary such that it accepts 5 trials.
- Patch the code such that it accepts any input as valid license.

Task 4: Feature Unlocking

Open `license_check` in Binary Ninja.

- Locate the hardcoded secret to unlock the premium feature.
- Patch the binary such that the feature is always unlocked.

Task 5: Embedded Firmware

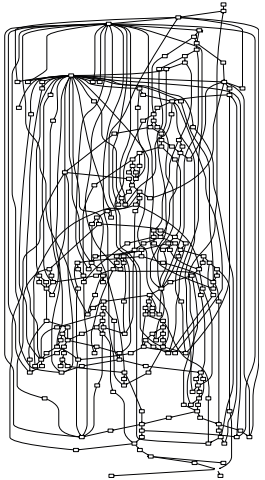
Open the embedded firmware `car_demo.elf` in Ghidra.

- Inspect the strings. What functionality might the firmware implement?
- The firmware hardcodes Wifi credentials (SSID & password). Find them.
- What is the IP address of the router?

Anti Rever Engineering

Code Obfuscation & Data Encoding

- increase **code complexity** to impede **reverse engineering** (code obfuscation)
- **hide keys and credentials** and decode them at **runtime**

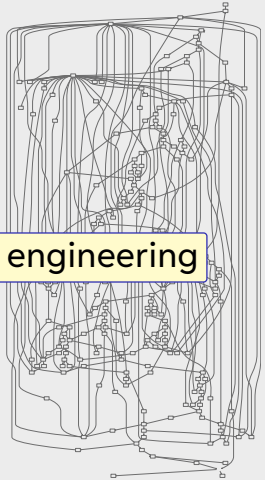


Code Obfuscation & Data Encoding

- increase **code complexity** to impede **reverse engineering** (code obfuscation)

passive protections impede reverse engineering

- **hide keys and credentials** and decode them at **runtime**



Anti-Debug & Anti-Tamper

- observe execution environment for debuggers (anti-debug)

```
if debugger_detected() {
    terminate()
}
```

- detect code modifications (patching) by code checksumming (anti-tamper)

```
if checksum(code) != 0xd75648 {
    terminate()
}
```

Anti-Debug & Anti-Tamper

- observe execution environment for debuggers (anti-debug)

```
if debugger_detected() {
    terminate()
}
```

runtime protections to prevent analysis & modifications

- detect code modifications (patching) by code checksumming (anti-tamper)

```
if checksum(code) != 0xd75648 {
    terminate()
}
```

Conclusion

Conclusion

- reverse engineering, machine code & assembler, tools
- hands-on sessions
- anti reverse engineering techniques

Try it yourself:

`https://github.com/emproof-com/workshop_software_reverse_engineering_escar24`

Tim Blazytko

`@mr_phrazer`

`https://www.emproof.com/`

tblazytko@emproof.com