

PV204 Security technologies



Rootkits, reverse engineering of binary applications

Petr Švenda  svenda@fi.muni.cz  [@rngsec](https://twitter.com/rngsec)

Centre for Research on Cryptography and Security, Masaryk University

CRCS

Centre for Research on
Cryptography and Security

What is planned for this lecture?

- Rootkits (and defences)
- Reverse engineering (of binary applications)

K. Thompson – Reflections on Trusting Trust

- Subverted C compiler (Turing Award Lecture, 1983)
 - Adds additional functionality for selected compiled programs
 - E.g., *login* cmd: log password or allow user with specific name
- Inspection of *login*'s source code will not reveal any issues
- Adds malicious functionality of compiler into binary of compiler compiled with already subverted compiler
 - Inspection of source code of compiler will not reveal any problem
- How can we detect modified *login* binary?
 - Expected hash, digital signatures, deterministic build...
 - What if signature verification tool is also modified?
- W32/Induc-A infected compiler for Delphi (2009)
 - Active at least a year before discovery

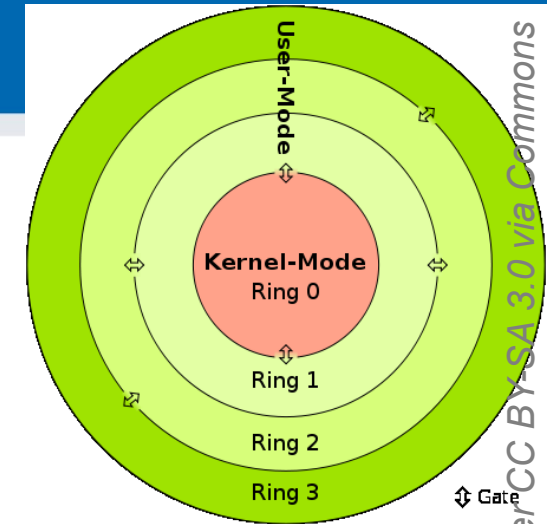
ROOTKITS

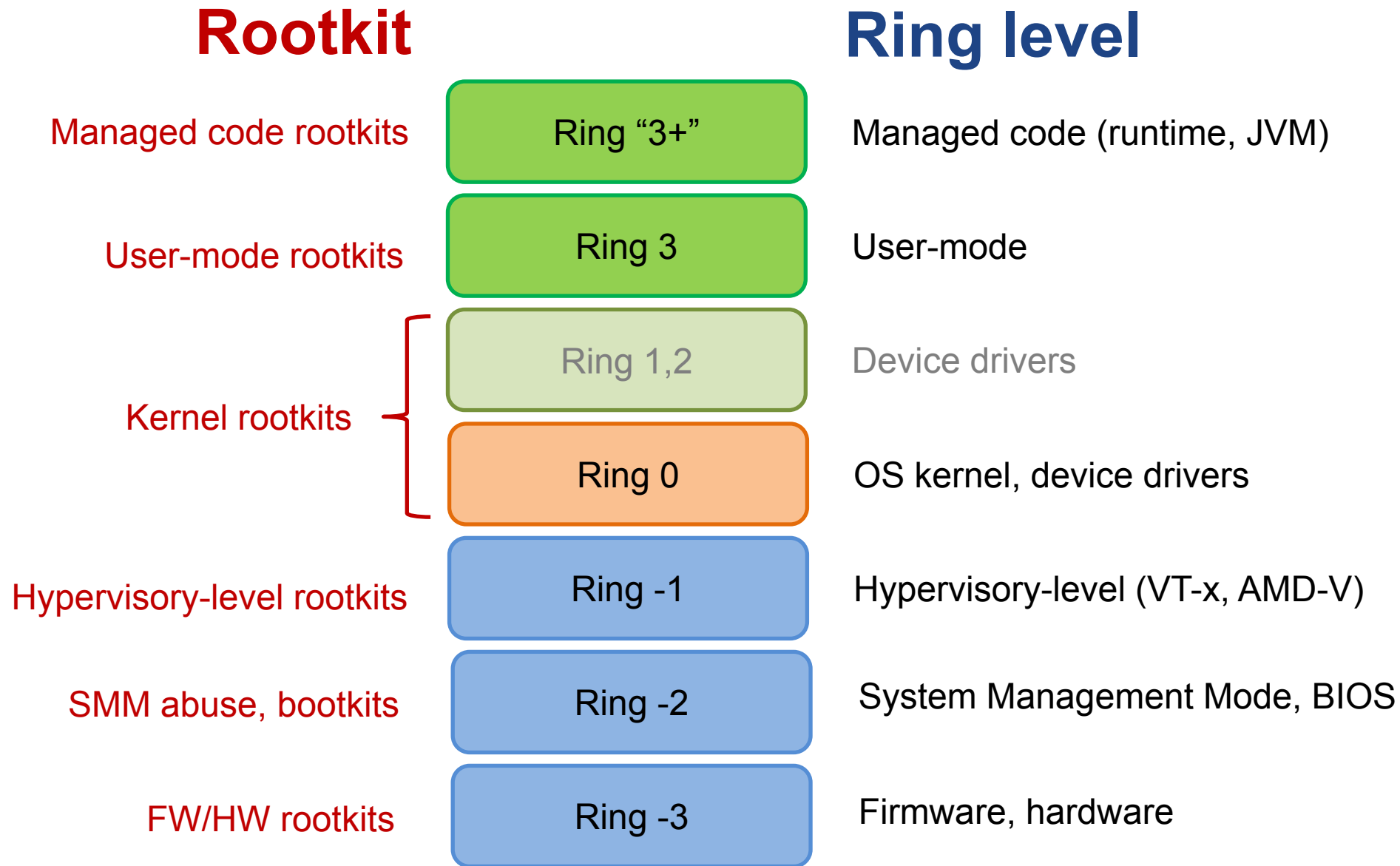
Rootkit definition

- Root-kit
 - *root* user *nix systems
 - *kit* set of tools to operate/execute commands
- Rootkit is piece or collection of software
 - Designed to enable access where it would be otherwise denied
 - Tries to hide(“cloak”) its presence in system
- Installed after obtaining privileged access
 - Privileged escalation, credentials compromise, physical access...
- Rootkit != exploit (rootkit usually installed after exploit)
- Rootkit is usually accompanied with additional payload
 - Payload does the actual (potentially malicious) work

Protection rings

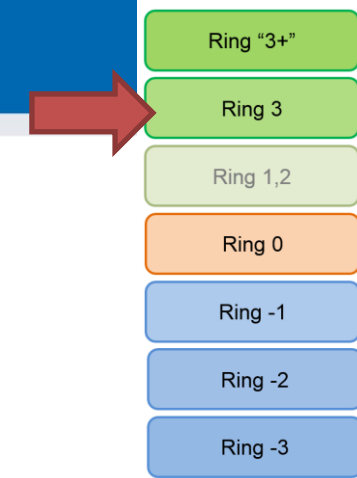
- Idea: introduce separate runtime levels
 - Crash in level X causes issue only in levels $\geq X$
 - Direct support provided by CPU architectures (0/3)
 - Instructions which can be executed only in given ring
- Ring 3: unprivileged user programs
- Ring 2/1: device drivers (currently sparsely used)
- Ring 0: kernel programs
- Performance penalty associated with ring switching
 - In practice, only 3 and 0 are commonly used
- 0-3 Captures only rings/levels starting with OS
 - Levels -1/-2/-3 introduced for layers below OS





Principal ways of detection of rootkits

1. Detection running inside system, same or higher level
 - Flaws in rootkit cloaking, use of some side-channel leakage (of rootkit)
2. Detection running inside system, lower level
 - Not controlled by rootkit, rootkit cannot cloak itself
3. Detection via (offline) image of system / memory
 - Rootkit is not running => cannot cloak itself



User-mode rootkits (Ring 3)

- Injects payload into other user applications
 - Injection of modified dlls (user app will use different CreateFile)
 - Modification of applications (modification of CreateFile)
- Interception of messages
 - RegisterWindowMessage()
- Function hooking
 - More generic hooks (SetWindowsHookEx()) – window manager
 - User application-specific hooks (plugins, example browser hook)
- File-system filters
 - Detect access to files by user application

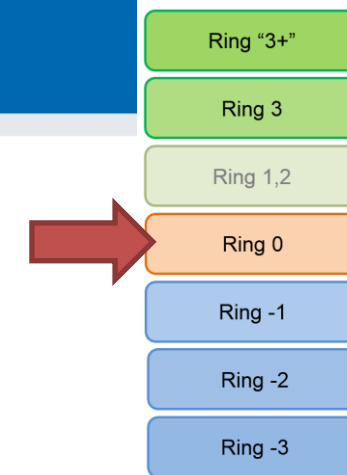


Managed code rootkits (MCR) (Ring 3)

- Ring 3 (level for runtime / VM)
- Targets runtime environments for interpreted code
 - .NET VM, Java VM and Dalvik runtime...
- Large attack surface for MCR
 - Attacking runtime class libraries
 - Attacking JIT compiler
 - Abusing runtime instrumentation features
 - Extending language with malware API
 - Object-oriented malware (inside OO runtime)
- E. Metula: Managed Code Rootkits (Syngress)

Kernel-mode rootkits (Ring 0)

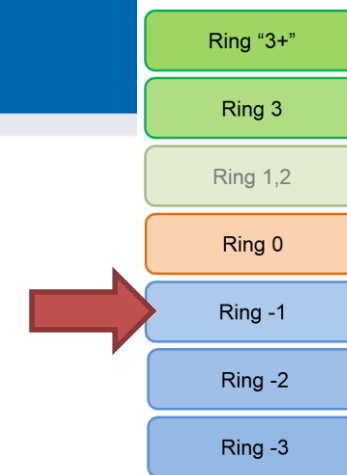
- Runs with highest system privileges
 - Usually device drivers and loadable modules
 - Device drivers in MS Windows
 - Loadable kernel modules in Linux
- Direct kernel object manipulation
 - Data structures like list of processes...
 - System Service Descriptor Table (SSDT) hook [Microsoft]
 - System call table hook [Linux]
- Operating system may require mandatory drivers signing
 - More difficult to insert malicious driver
 - Still possible (compromised private keys: Stuxnet & Realtek's keys)



ROOTKITS BELOW OS LEVEL

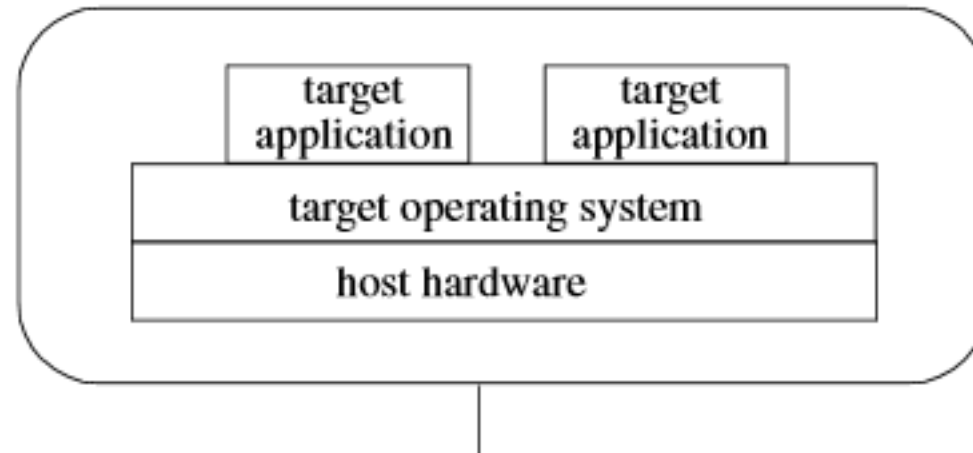
Hypervisory-level rootkits (Ring -1)

- Virtual-machine based rootkit (VMBR)
 - Type II hypervisors (VM on ordinary OS host)
- Based on CPU hardware virtualization features
 - Intel VT or AMD-V
- Rootkit hosts original system as virtual machine
 - And intercepts all relevant hardware calls
- Examples: SubVirt, BluePill (AMD-V, Intel VT-x)



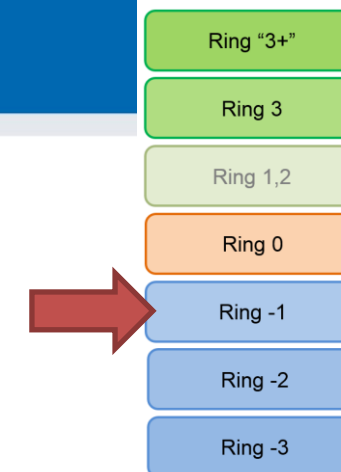
Hypervisory-level rootkits (Ring -1)

Before infection



King et al: SubVirt: Implementing malware with virtual machines

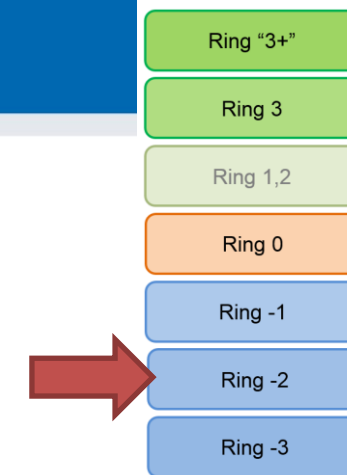
Defense against hypervisory-level rootkits



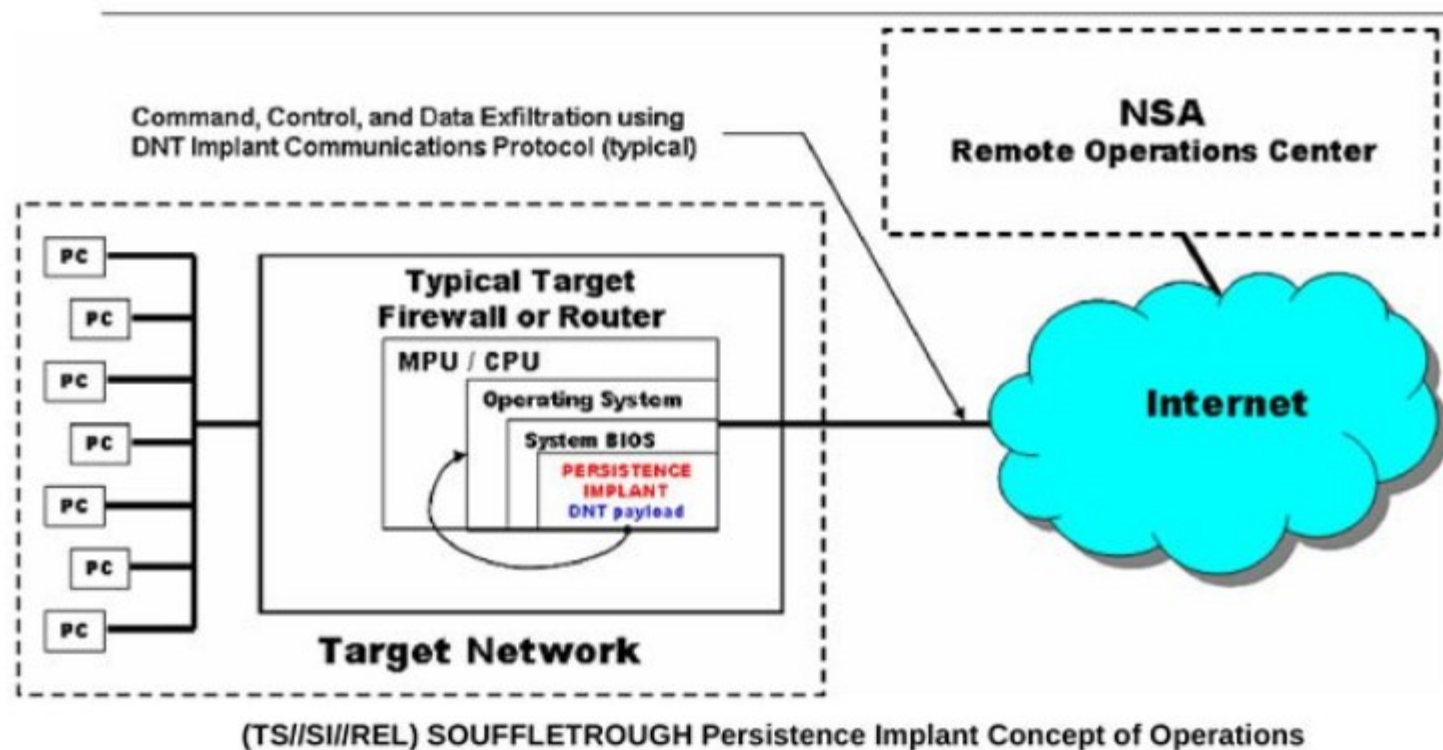
- Run detection/prevention on lower level
- Detect by timing differences of operations
 - System is emulated => side-channel info (timings...)
- Read and analyze HDD physical memory
 - After physical removal from (infected) computer
- Boot from safe medium (CD, USB, network boot)
 - inspect before VMBR loads
 - But VMBR can emulate shutdown / reboot
 - Physical power unplug recommended
- Trusted boot (based on TPM, lecture 07)

System Management Mode abuse (R.-2)

- System Management Mode (SMM)
 - x86 feature since Intel 386, all normal execution is suspended
 - Used for power management, memory errors, hardware-assisted debugger...
 - High-privilege mode (Ring -2)
- SMM entered via system management interrupt (SMI)
 - System cannot override or disable the SMI
- Target for rootkits
 - Modify memory, loaders, MBR...



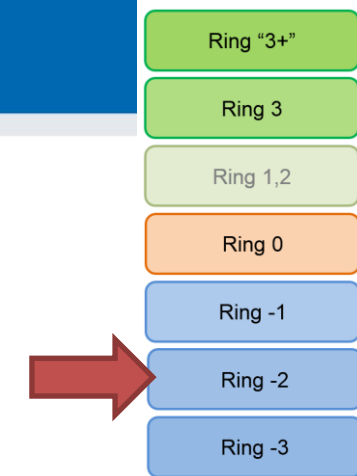
SMM Example: SOUFFLETROUGH implant



- https://en.wikipedia.org/wiki/NSA_ANT_catalog
- <http://leaksource.info/2013/12/30/nsas-ant-division-catalog-of-exploits-for-nearly-every-major-software-hardware-firmware/>

Bootkit rootkits (Ring -2)

- Bootkit = Rootkit + Boot capability
- Infect startup code
 - Master Boot Record (MBR)
 - Volume Boot Record (VBR)
 - Boot sector, BIOS routines...
- “Evil maid” attack
 - Can be used to attack full disk encryption
 - Assumption: user will left device physically unattended
 - Legitimate bootloader replaced (+ key capture)



Full-disk encryption compromise

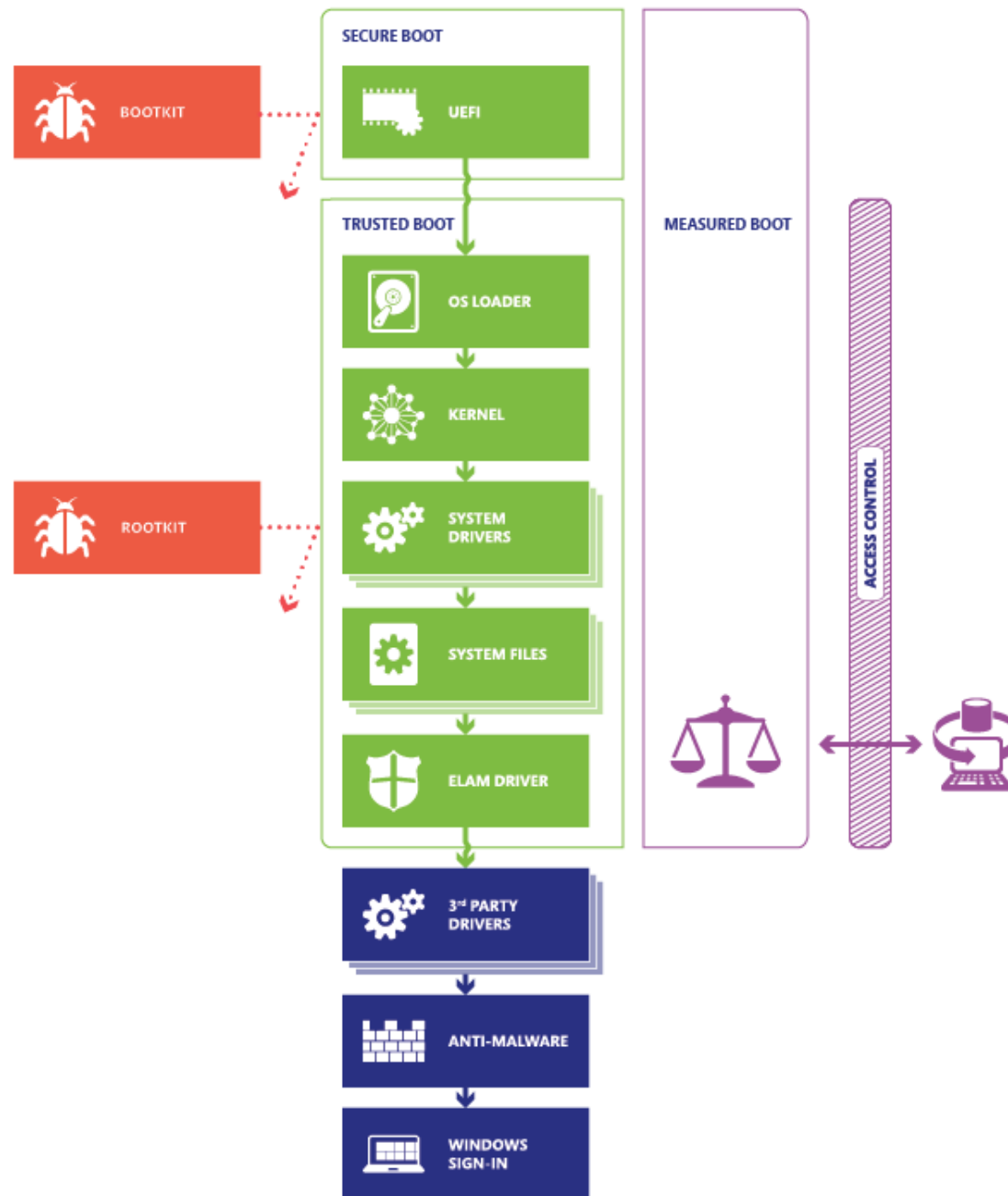
1. Full-disk encryption used to encrypt all data
 2. Laptop powered down to prevent Coldboot or FireWire-based attacks (read key from memory)
 3. Laptop left unattended (“Evil maid” enters)
 - USB used to read part of first sector of disk
 - If TrueCrypt/Bitlocker loader, then insert malicious bootloader
 4. User is prompted with forged bootloader
 - Password is stored
- How to transfer saved password / data to attacker?
 - Second visit of Evil maid



<http://theinvisiblethings.blogspot.co.uk/2009/10/evil-maid-goes-after-truecrypt.html>

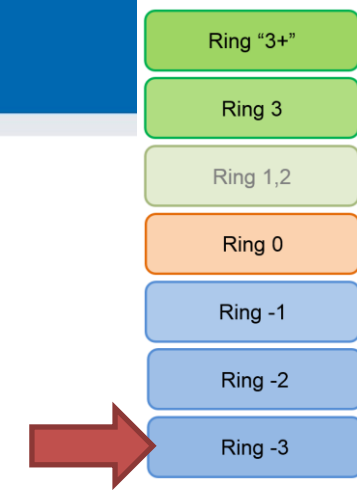
Bootkit defenses

- Prevention of physical access
 - Problematic for portable devices
- Trusted boot (static vs. dynamic root of trust)
 - Refer to Lecture 07 (Trusted boot)
 - But bootloader must authenticate itself to user
 - E.g., present image encrypted by key stored in TPM
 - Before user enters its password
- Defense by external verification of bootloader integrity
 - verify relevant unencrypted parts of disk (external USB)



Firmware / hardware rootkits (Ring -3)

- Persistent malware image in hardware
 - Network card, router, hard drive...
- Can run even after removal of device from target computer
 - Once device is powered again



LEGITIMATE USES

Legitimate uses of rootkits

- To whom is legitimacy measured?
- Hide true nature of network “honeypots”
- Protection of AV software against termination of inspected malware
- Anti-theft protections
- Digital rights management?

Sony BMG Extended copy protection

- Rootkit developed for (and approved) by Sony
 - Intended to limit possibility for disk copy
 - Users were not notified (silently installed after CD insert)
 - Digital rights management for Sony
 - To hide itself, any file starting with \$sys\$ was hidden
- Detected by M. Russinovich's RootkitRevealer
 - After public disclose, other malware started to hide itself by naming its files as \$sys\$ (user was already "infected")
- Sony released patch for removal (web-based uninstaller)
 - Even more serious flaw introduced (any visited page can install and run program)
 - Resulted in class-action lawsuit against Sony BMG

REVERSE ENGINEERING

Reverse engineering

- A process of knowledge or design extraction from final product (usually man-made)
- Engineering:
 - Mental model → blueprints/source-code → product/binary
- Reverse engineering (back engineering):
 - From product back to knowledge or design
 - Blueprints/source-code might be also recreated
- Not necessary/possible to perfectly recreate design
 - Engineering might be loose transformation
 - Back engineering might not be perfect/complete

Reverse engineering is general process
We will focus on software binaries only

Reverse engineering - legal issues

- Reverse engineering is legal when
 - Own binary without documentation
 - Anti-virus research, Forensics...
 - Interoperability, Fair use, education
- Problem with some copyright laws
 - not only selling circumvented content, but also attempt to circumvent is illegal (USA's DMCA)
- EFF Coders' Rights Project Reverse Engineering FAQ
 - Legal doctrines, Risky aspects, Selected decisions
 - <https://www.eff.org/issues/coders/reverse-engineering-faq>

How to start reverse engineering

1. Learn basic concepts (compilers, memory, OS...)
2. See how source-code translates into binary
3. Try tools on simple examples (own code, tuts)
4. Utilize other knowledge (communication logs...)
5. Have fun! 😊

Basics

- Debugger vs. debugger with binary modification capabilities
 - E.g., Visual Studio vs. OllyDbg
- Disassembler vs. debugger
 - Static vs. dynamic code analysis
- Disassembler vs. decompiler
 - Native code → assembler → source code
- Native code vs. bytecode
 - Different instruction set, different execution model
- Registry-based vs. stack-based execution

Mixed source code/assembler in IDE

- Most current IDE supports mixed source code/assembler instructions mode (Visual Studio, QT Creator...)
 - Mode is usually available only during a debugging
 - Write simple code (e.g., if then else condition), insert breakpoint and start debugging
- Switch to mixed mode
 - Visual Studio→RClick→Go to disassembly
 - QTCreator→Debug→Operate by Instruction
- Easy way to learn how particular source code is translated into assembler code


```

#include <stdio.h>
int main() {
    FILE* file = NULL;
    file = fopen("values.txt", "r");

    if (file) {
        int value1 = 0;
        int value2 = 0;
        fscanf(file, "%d", &value1);
        fscanf(file, "%d", &value2);

        value1 = value1 + value2;

        printf("Result: %d", value1);
    }
    fclose(file);
}

```

Original C source code

```

00401340  CS      RETN
00401341  90      NOP
00401342  90      NOP
00401343  90      NOP
00401344  55      PUSH EBP
00401345  89E5   MOV EBP,ESP
00401347  83E4 F0 AND ESP,FFFFFF0
0040134A  83EC 20 SUB ESP,20
0040134D  E8 CE060000 CALL Test_C.00401A20
00401352  C74424 1C 000 MOV DWORD PTR SS:[ESP+1C],0
0040135A  C74424 04 302 MOV DWORD PTR SS:[ESP+4],Test_C.00402030
00401362  C70424 322040 MOV DWORD PTR SS:[ESP],Test_C.00402032
00401369  E8 22090000 CALL <JMP.&nsvcrt.fopen>
0040136E  894424 1C MOV DWORD PTR SS:[ESP+1C],EAX
00401372  837C24 1C 00 CMP DWORD PTR SS:[ESP+1C],0
00401377  74 6B JE SHORT Test_C.004013E4
00401379  C74424 18 000 MOV DWORD PTR SS:[ESP+18],0
00401381  C74424 14 000 MOV DWORD PTR SS:[ESP+14],0
00401389  8D4424 18 LEA EAX,DWORD PTR SS:[ESP+18]
0040138D  894424 08 MOV DWORD PTR SS:[ESP+8],EAX
00401391  C74424 04 302 MOV DWORD PTR SS:[ESP+4],Test_C.00402030
00401399  8B4424 1C MOV EAX,DWORD PTR SS:[ESP+1C]
0040139D  890424 MOV DWORD PTR SS:[ESP],EAX
004013A0  E8 F3000000 CALL <JMP.&nsvcrt.fscanf>
004013A5  8D4424 14 LEA EAX,DWORD PTR SS:[ESP+14]
004013A9  894424 08 MOV DWORD PTR SS:[ESP+8],EAX
004013AD  C74424 04 302 MOV DWORD PTR SS:[ESP+4],Test_C.00402030
004013B5  8B4424 1C MOV EAX,DWORD PTR SS:[ESP+1C]
004013B9  890424 MOV DWORD PTR SS:[ESP],EAX
004013BC  E8 D7000000 CALL <JMP.&nsvcrt.fscanf>
004013C1  8B5424 18 MOV EDX,DWORD PTR SS:[ESP+18]
004013C5  8B4424 14 MOV EAX,DWORD PTR SS:[ESP+14]
004013C9  8D0402 LEA EAX,DWORD PTR DS:[EDX+EAX]
004013CC  8B4424 18 MOV EAX,DWORD PTR SS:[ESP+18]

```

Dump of assembler code for function main:

```

2      int main() {
0x00401344 <+0>:      push    %ebp
0x00401345 <+1>:      mov     %esp,%ebp
0x00401347 <+3>:      and    $0xffffffff,%esp
0x0040134a <+6>:      sub    $0x20,%esp
0x0040134d <+9>:      call   0x401a20 <__main>

3      FILE* file = NULL;
0x00401352 <+14>:     movl   $0x0,0x1c(%esp)

4      file = fopen("values.txt", "r");
0x0040135a <+22>:     movl   $0x402030,0x4(%esp)
0x00401362 <+30>:     movl   $0x402032,(%esp)
0x00401369 <+37>:     call  0x401c90 <fopen>
0x0040136e <+42>:     mov    %eax,0x1c(%esp)

...

17     }
0x004013f5 <+177>:    leave
0x004013f6 <+178>:    ret

```

End of assembler dump.

Most common instructions/structures

- Most common ASM instructions
 - Load/Store from to registers: MOV, LEA
 - Arithmetic: ADD, INC...
 - Relational: CMP, TEST
 - Jumps: JMP, J*
 - Functions: CALL, RET
- Example of typical structures (C→ASM)
 - Conditional jump, `for` loop, function call...
 - Familiarize via mixed source code/assembler in IDE
 - Be aware of debug/release differences

Compilation to bytecode (Java, C#)

- Source code compiled into intermediate bytecode
 - Java bytecode, .NET CLI ...
- Intermediate code interpreted by virtual machine
- Just-in-time compilation
 - Intermediate code is compiled by VM into native code
 - Improve performance significantly
 - Relevant for dynamic analysis, not for static analysis
- Usually easier to understand than assembler code

REGISTRY VS. STACK-BASED EXECUTION

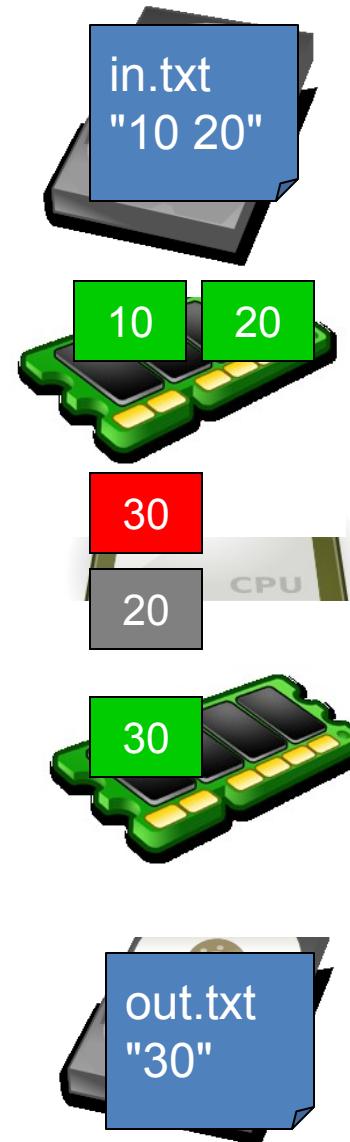
Registry-based execution

1. Values loaded (`mov`) from RAM to CPU registers
 2. CPU operation (`add`, `inc`, `test...`) is executed
 3. Resulting value is stored back (`mov`) to RAM
- Name of the registers
 - EAX 32bit, AX 16bit, AH/AL 8bit
 - EIP ... next address to execute (instruction pointer)
 - EBX ... usually loop counter
 - Registers
 - Z – zero flag, C – carry flag, S – sign flag...

Add two numbers from file (HDD)

1. Read values from HDD into RAM memory
 - `fscanf(file, "%d", &value);`
2. Move value from RAM memory to CPU registry
 - `MOV 0x48(%esp), %eax`
 - `MOV 0x44(%esp), %edx`
3. Execute CPU instruction (e.g., ADD)
 - `ADD %edx, %eax`
4. Transfer result from CPU register to RAM memory
 - `MOV %eax, 0x48(%esp)`
5. Save result from RAM memory to file
 - `fprintf(file, "%d", value);`

value = value + value2;



Stack-based execution

- Bytecode contains sequence of operations
- Bytecode contains constants
- All intermediate values stored on stack
- Interpret:
 1. Reads next operation from bytecode
 2. Pop operand(s) for next operation from top of stack
 3. Executes operation
 4. Push result of operation on top of stack
- No registers are used
 - all operands for current operation at the top of the stack

Example: JavaCard bytecode

```
// ENCRYPT INCOMING BUFFER
void Encrypt(APDU apdu) {
    byte[]    apdubuf = apdu.getBuffer();
    short     dataLen = apdu.setIncomingAndReceive();
    short     i;

    // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
    if ((dataLen % 8) != 0)
        ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

    // ENCRYPT INCOMING BUFFER
    m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen,
                            m_ramArray, (short) 0);

    // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
    Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf,
                            ISO7816.OFFSET_CDATA, dataLen);

    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
}
```

Original JavaCard source code

```
.method Encrypt(Ljavacard/framework/APDU;)V 129 {
    .stack 6;
    .locals 3;
    .descriptor Ljavacard/framework/APDU;    0.10;
L0:    aload_1;
        invokevirtual 30;
        astore_2;
        aload_1;
        invokevirtual 42;
        sstore_3;
        sload_3;
        bspush 8;
        srem;
        ifeq L2;
L1:    sspush 26384;
        invokestatic 41;
        goto L2;
L2:    getfield_a_this 1;
        aload_2;
        sconst_5;
        sload_3;
        getfield_a_this 10;
        sconst_0;
        invokevirtual 43;
        pop;
        getfield_a_this 10;
        sconst_0;
        aload_2;
        sconst_5;
        sload_3;
        invokestatic 44;
        pop;
        aload_1;
        sconst_5;
        sload_3;
        invokevirtual 45;
        return;
}
```

}

Resulting JavaCard bytecode

Recovering information from binary executables

DISASSEMBLING

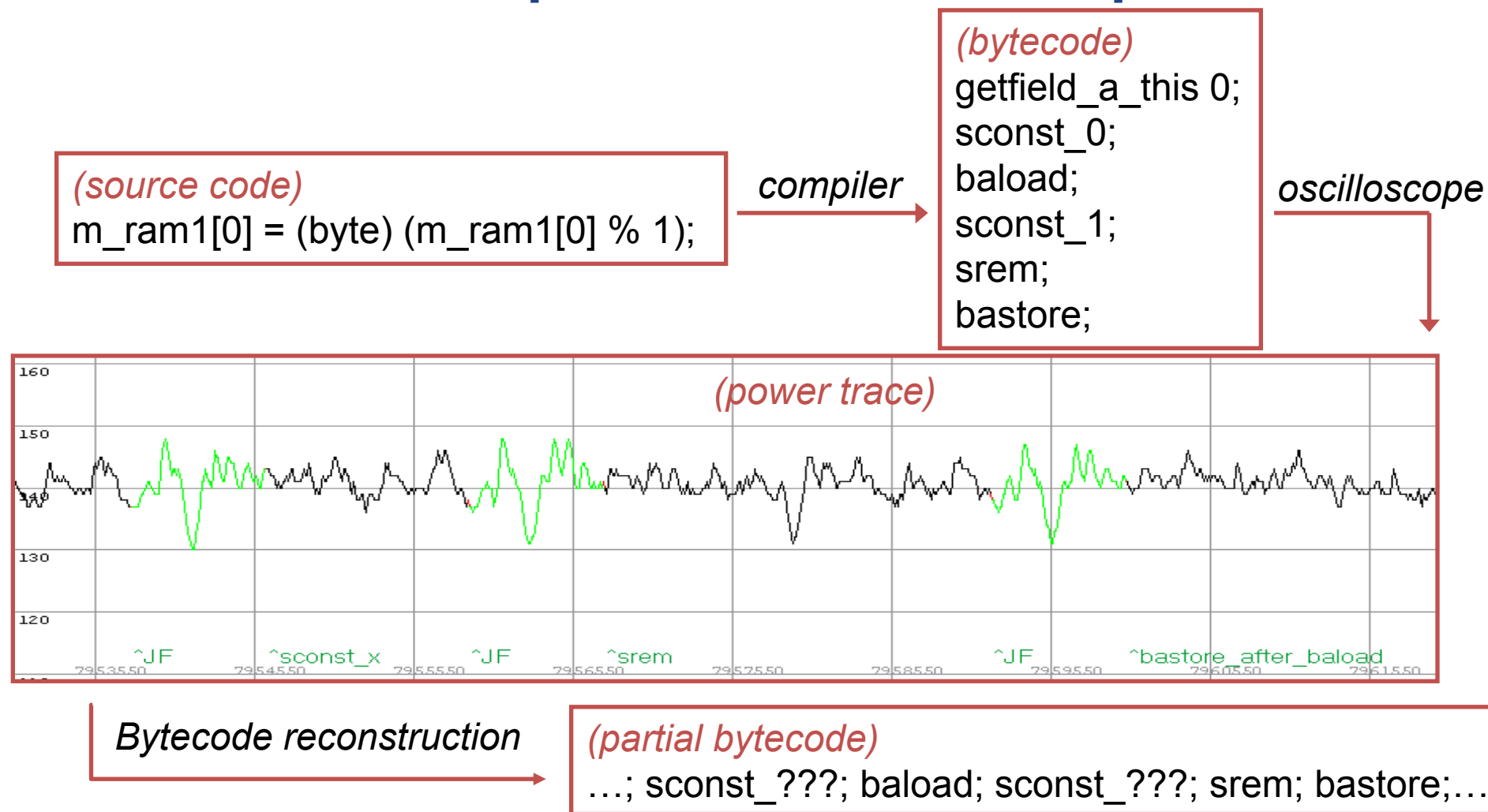
Disassembling of native binaries

- Reversing process of compilation
 - Back from native code to ASM
- Compilation/assembly is loose process:
 - Variable/function names
 - Unused structures
 - Performance optimization applied during compilation
- Wide range of native platforms
 - Differences in support and performance of disassemblers
- Bytecode is already on the level of “disassembled” binaries (usually easier to understand)

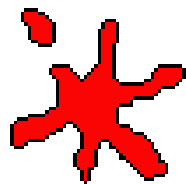
Structured code vs. sequence of executed ops

1. Structured code contains code for all branches
 - runnable binary/bytecode
 - Information loss in compiled binary
 - Stripped metadata and debugging symbols
 - Compiler optimizations
2. Sequence of executed instructions only from the branches taken
 - E.g., power analysis of smart card with recognized operations
 - If loop was executed, then only linear sequence of instructions is observed corresponding to the number of loop iterations

Structured code vs. sequence of executed ops



Tool: OllyDbg



- Free disassembler and binary debugger
 - Works with Windows 32b binaries only
 - OllyDbg 64b version in development (but last update in 2014 ☹️)
- Easy to start with, many tutorials
- Designed to make changes in binary easy
 - Change of jumps/data (valid PE is recreated)
- <http://www.ollydbg.de/>



IDA Pro
by Ilfak Guilfanov

Tool: IDA Pro

- Interactive Disassembler is legendary full-fledged disassembler with ability to disassemble many different platforms
- Free version available for non-commercial uses
 - <http://www.hex-rays.com/idapro/idadownfreeware.htm>
- Free version disassemble only Windows binaries
- Very nice visualization and debugger feature (similar as OllyDbg)



Tool: Online disassembler (ODA)

- <https://www.onlinedisassembler.com/odaweb/>

The screenshot shows the ODA web interface. At the top, there is a navigation bar with the ODA logo, a 'Share' button, and a menu with 'File', 'Edit', 'Examples', 'Help', 'Blog', and 'Contact Us!'. On the right side of the navigation bar are 'Sign in' and 'Sign Up' buttons.

The main content area is divided into two columns. The left column, titled 'Live View', contains instructions: 'Set the platform below. Then watch the disassembly window update as you type hex bytes in the text area. You can also upload an ELF, PE, COFF, Mach-O, or other executable file from the File menu.' Below this is a red button labeled 'Platform: i386' and a text input field containing the hex string: '55 31 D2 89 E5 8B 45 08 56 8B 75 0C 53 8D 58 FF 0F B6 0C 16 88 4C 13 01 83 C2 01 84 C9 75 F1 5B 5E 5D C3'. A blue arrow points from this input field to the disassembly window.

The right column, titled 'Disassembly', has tabs for 'Disassembly', 'Hex', 'Sections', and 'File Info'. The 'Disassembly' tab is active, showing a list of assembly instructions with their addresses and hex values. The instructions are:

```

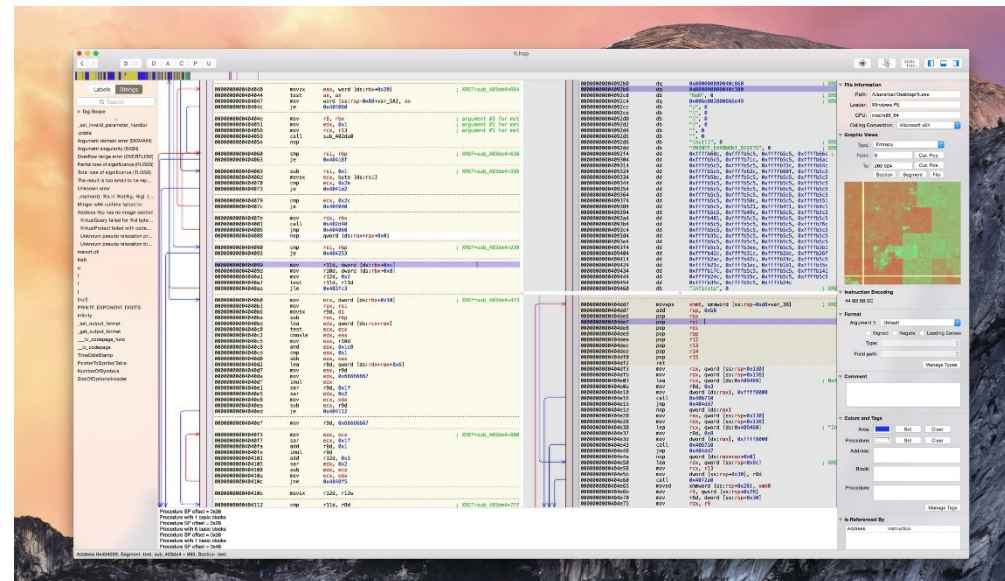
.data:00000000 55      push ebp                                ;press the ';' button to make a comment
.data:00000001 31d2    xor edx,edx                             ;you can also click the right mouse butt
.data:00000003 89e5    mov ebp,esp
.data:00000005 8b4508  mov eax,DWORD PTR [ebp+0x8]
.data:00000008 56      push esi
.data:00000009 8b750c  mov esi,DWORD PTR [ebp+0xc]           ;char* src = arg[1]
.data:0000000c 53      push ebx
.data:0000000d 8d58ff  lea ebx,[eax-0x1]                     ;char* dst = arg[0]
.data:00000010
.data:00000010 loop:
.data:00000010 0fb60c16 movzx ecx,BYTE PTR [esi+edx*1]         ;char c = src[i]
.data:00000014 884c1301 mov BYTE PTR [ebx+edx*1+0x1],cl       ;dst[i] = c
.data:00000018 83c201  add edx,0x1                           ;i++
.data:0000001b 84c9    test cl,cl                             ;while (c != 0)
.data:0000001d 75f1    jne loop
.data:0000001f 5b     pop ebx
.data:00000020 5e     pop esi
.data:00000021 5d     pop ebp
.data:00000022 c3     ret
  
```

At the bottom of the interface, there is a status bar showing '[normal] ONLINE DISASSEMBLER 4EVER!' on the left, 'strcpy (x86) : i386 (70 bytes)' in the middle, and '0x0' on the right.



Tool: Hopper disassembler and debugger

- Linux and OS X reverse engineering tool
 - Older version supported Windows, but not anymore
- <http://www.hopperapp.com>
- Additional support for Objective-C



Radare

- <https://www.radare.org/r/>
- Multiplatform
- Very good support for multiple file formats and platforms
- (Unfair 😊) comparison with other tools
 - <https://www.radare.org/r/cmp.html>

The screenshot displays the Radare2 interface with the following components:

- Functions List:** A list of functions including entry0 and various fcn. entries.
- Disassembler:** Shows assembly code for the selected function. The instruction at address 0x4047bf is highlighted in green: `mov rax, qword [rsp + 0x40]`. Other instructions include `mov rax, qword [rip + 0x218304]`, `mov rdi, qword [rsp + 0x28]`, `lea rsi, qword [rsp + 0x38]`, `mov edx, 1`, `mov rcx, r13`, `add qword [rsp + 0x38], 1`, `mov qword [rip + 0x2182e5], r13`, `mov qword [r13 + 0x20], rax`, `mov qword [r13 + 8], rax`, `call 0x404a70`, `cmp al, 1`, `sbb edx, edx`, `and edx, 2`, `add edx, 3`, `jmp 0x404362`, `; JMP XREF from 0x404763`, `mov rax, qword [rsp + 0x40]`, `movsxd rcx, r14d`, `mov rdi, qword [rsp + 0x28]`, `shl rcx`, `lea rsi, qword [rsp + 0x38]`, `xor edx, edx`, `add rcx, 0x61bc80`, `mov qword [rcx + 8], rax`, `call 0x404a70`, `xor edx, edx`, `test al, al`, `jne 0x40436b`, `; JMP XREF from 0x404775`, `lea rdi, qword [rsp + 0xf0]`, `call 0x40eaa0`, `xor edi, edi`, and `mov r14, rax`.
- Information Panel:** Displays file metadata such as `file /bin/ls`, `type EXEC (Executabl`, `pic false`, `canary true`, `nx true`, `crypto false`, `va true`, `root elf`, `class ELF64`, `lang c`, `arch x86`, `bits 64`, `machine AMD x86-64 arch`, `os linux`, `subsys linux`, `endian little`, `strip true`, `static false`, `linenum false`, `lsyms false`, `relocs false`, `rpath NONE`, `type EXEC (Executabl`, `os linux`, `arch AMD x86-64 arch`, `bits 64`, `endian little`, `file /bin/ls`, `fd 6`, `size 0x1c6f8`, `mode r--`.
- Registers:** A table showing the current state of registers:

| | | | |
|-------|------------|-----|------------|
| > ar= | | | |
| r15 | 0x00000000 | r13 | 0x00000000 |
| r12 | 0x00000000 | rbp | 0x00000000 |
| r11 | 0x00000000 | r10 | 0x00000000 |
| r8 | 0x00000000 | rax | 0x00000000 |
| rdx | 0x00000000 | rsi | 0x00000000 |
| orax | 0x00000000 | rip | 0x00000000 |
| rsp | 0x00000000 | | |

Decompilation

- Native code decompilation
 - Decompiler produces source code from binary/ASM/bytecode code
 - Decompiler needs to do disassembling first and then try to create code that will in turn produce binary code you have at the beginning
 - Resulting code will NOT contain information removed during compilation (comments, function names, formatting...)
- Bytecode decompilation
 - usually much easier (more information preserved)
 - Mapping between source code and bytecode is less ambiguous
 - Compilation of decompiled bytecode produces similar bytecode

Decompiler tools

- C/C++
 - IDA
 - REC Studio 4.0, <http://www.backerstreet.com/rec/rec.htm>
 - Retargetable Decompiler, <https://retdec.com/>
 - Ghidra – diassembler by NSA <https://github.com/NationalSecurityAgency/ghidra>
- Java bytecode
 - DJ Java Decompiler, <http://neshkov.com/dj.html>
 - Java Decompiler, <http://jd.benow.ca/>
- .Net bytecode
 - dotPeek, <https://www.jetbrains.com/decompiler/>
 - ILSpy, <http://ilspy.net/>

Summary

- Several levels where rootkit can be placed (CPU rings)
- Rootkits cloaks itself and run malicious functionality
 - Detection on higher/same level difficult (but possible if cloaking is not perfect)
 - Try to detect on lower level (root is not “running”)
 - Trusted boot (TPMs...) attempts to prevent/detect rootkit execution
- Reverse engineering of binaries
 - Compilation is lossy process (debug symbols, optimizations...)
 - Different platforms have different binaries (registry/stack-based execution, instructions, function calling conventions...)
 - Disassembling: binary -> assembler
 - Decompilation: assembler -> source code