1 Reverse Engineering

- Software-Wartung in gewachsenen IT-Systemen
- Herstellung von Kompatibilität
- Malware-Analyse: Verhaltensanalyse
- Exploitation: Schwachstellen aufspüren
- Cracking

2 Assembler

Klassifizierung von Programmiersprachen:

- 1st Generation: Maschinensprache (Opcodes: Binärdarstellung eines Maschinenbefehls)
- 2nd Generation: Assemblersprache (Mnemonics: menschenlesbares Symbol eines Maschinenbefehls in ASM)
- 3rd Generation: platform-unabhängige Hochsprache (bspw. C)

Register:

- Daten-Register: EAX (Accumulator), EBX (Base), ECX (Counter), EDX (Data)
- Index-Register: ESI (Source), EDI (Destination)
- Zeiger-Register: EBP (Base), ESP (Stack), EIP (Instruction)
- Segment-Register: CS, SS, DS, ES, FS, GS
- Status-Register: (E)FLAGS

CF	Carry Flag	1 falls letzte Op VZ-losen Wertebereich überschritten
PF	Parity Flag	1 falls letzte Op gerade Anz. von 1en im niederwertigen Byte erzeugt (VZ-behaftet)
AF	Auxilliary CF	1 falls letzte Op Überlauf im unteren Halbbyte hatte
ZF	Zero Flag	1 falls Ergebnis der letzten Op 0 war
SF	Sign Flag	1 falls Ergebnis der letzten Op negativ war
DF	Direction Flag	steuert in welcher Richtung Strings bearbeitet werden
OF	Overflow Flag	1 falls letzte Operation vorzeichenbehafteten Wertebereich überschritten hat

x86 Befehle:

- 1 Mnemonic gefolgt von 0 - 3 Operanden
- bei Befehlen mit 2 Operanden: einer zugleich Zieloperand
- nur eine Speicherreferenz pro Befehl
- Offset-Adressierung:
  Offset = Base + (Index * Scale) + Displacement
  Base: Register (EAX,EBX,ECX,EDX,ESP,EBP,ESI oder EDI)
  Index: Register (EAX,EBX,ECX,EDX,EBP,ESI oder EDI)
  Scale = 1,2,4,8, Displacement = 8 oder 32-Bit Wert
- ADD, SUB, ADC, SBB, INC, DEC, MUL, DIV, ...
  MUL,DIV: EDX:EAX wird als 64-Bit Register interpretiert
- CMP op1, op2: SUB, mit op1 und op2 Quelloperanden
  - ZF falls op1 == op2
  - SF falls op1 < op2
  - CF bei VZ-losem, OF bei VZ-behaftetem Überlauf
- TEST op1, op2: AND, mit op1 und op2 Quelloperanden
  - ZF falls (op1 & op2) == 0
  - SF := MSB von (op1 & op2)
- PUSH, POP, CALL, RET, PUSHF, POPF, PUSHAD, POPAD

VZ-los:			VZ-behaftet:		
JA (JNBE)	CF = 0 und ZF = 0	op1 > op2			
JAE (JNB)	CF = 0	op1 ≥ op2	JG (JNLE)	ZF = 0 und SF = OF	op1 > op2
JB (JNAE)	CF = 1	op1 < op2	JGE (JNL)	SF = OF	op1 ≥ op2
JBE (JNA)	CF = 1 oder ZF = 1	op1 ≤ op2	JL (JNGE)	SF ! = OF	op1 < op2
JE	ZF = 1	op1 == op2	JLE (JNG)	ZF = 1 und SF ! = OF	op1 > op2
JNE	ZF = 0	op1 ! = op2			

Unterprogrammaufrufe:

PUSH op	SUB ESP, 4; MOV [ESP], op	Speicher op auf Stack
POP reg	MOV reg, [ESP]; ADD ESP, 4	Lese TOS nach reg
CALL op	PUSH EIP; JMP op	Unterprogrammaufruf
RET [op]	POP EIP; [ADD ESP, op]	Rücksprung

Stack:

- ESP = Zeiger auf oberstes ELement (TOS)
- EBP = Zeiger auf Stack-Frame

Prolog: `push ebp; mov ebp, esp; sub esp, 0x??`
Epilog: `leave; ret`

2.1 Intel vs. AT&T Syntax

Intel Syntax	AT&T
Zieloperand links	Zieloperand rechts
Register/Konstanten ohne Präfix	Präfix: Konstanten = $, Register = %
Operandengröße über BYTE[], WORD[], ...	Operandengröße über Suffix b, w, ...
Effektive Adresse:	Effektive Adresse:
`[base+idx*scale+displace]`	`DISP(BASE,IDX,SCALE)`

2.2 CISC Befehle

LOOP adr	ECX–, Sprung falls ECX != 0
LOOPE adr	ECX–, Sprung falls ECX != 0 und ZF = 1
LOOPNE adr	ECX–, Sprung falls ECX != 0 und ZF = 0

Wiederholungspräfixe:
REP (ECX = 0), REPE/REPZ (ECX = 0 / ZF = 0), REPNE/REPNZ (ECX = 0 or ZF = 1)

Zeichenkettenbefehle (Suffix B,W,D):

MOVS	Kopiere Byte/Word/Dword von [ESI] nach [EDI]
LODS	Kopiere AL/AX/EAX nach [EDI]
STOS	Kopiere [EDI] nach AL/AX/EAX
CMPS	Vergleiche Byte/Word/Dword an [ESI] und [EDI] und setze Status-Flags
SCAS	Vergleiche AL/AX/EAX mit Byte/Word/Dword an [EDI] und setze Status-Flags

2.3 Erweiterungen

SIMD:
Multi Media Extensions (MMX): acht 64-Bit Register MM0 bis MM7 (überlappt mit 80-Bit FPU Registern)
Streaming SIMD Extensions (SSE): eigener Registersatz mit acht 128-bit Registern
Advanced Vector Extensions (AVX): mehr größere Register!

AMD64 Registersatz:
Erweiterung der bisherigen Register auf 64-bit + 8 neue 64-bit Register
neue Zugriffsmethoden auf Teilregister (alle Register mit 8/16/32/64-Bit ansprechbar)

2.4 Aufrufkonventionen

- Wo werden Parameter in welcher Reihenfolge abgelegt?
- Wer bereinigt den Stack von Parametern?
- Wie wird der Rückgabewert übergeben?

2.4.1 IA-32

- cdecl: Standard für C (bspw. GCC)
  Parameter auf Stack, die letzten Parameter zuerst auf den Stack, Caller-Cleanup
  Return-Value: EAX, Caller-Saved: EAX, ECX, EDX, Callee-Saved: EBX, ESI, EDI, EBP, ESP
- stdcall: Standard für Win32 API
  Parameter auf Stack, die letzten Parameter zuerst auf den Stack, Callee-Cleanup
  Return-Value: EAX, Caller-Saved: EAX, ECX, EDX, Callee-Saved: EBX, ESI, EDI, EBP, ESP

- fastcall: Parameter soweit möglich in Registern übergeben
  Parameter 1: ECX, Parameter 2: EDX, Rest in umgekehrter Reihenfolge auf Stack, Callee-Cleanup
- weitere: thiscall, Pascal, optlink (IBM), ...

2.4.2 AMD64

- System V/AMD64 ABI: Caller-Cleanup, Parameterübergabe in RDI, RSI, RDX, RCX, R8, R9, dann Stack
- Microsoft x64: wie SystemV, aber Parameterübergabe in Registern (RCX, RDX, R8, R9), dann Stack

AMD64 ABI Red Zone:
- 128-byte Stackplatz, der genutzt werden darf, ohne einen Stack-Frame dafür anlegen zu müssen
- nur für Leaf-Funktionen, da Unterprogrammaufrufe die Red-Zone nicht überspringen

2.5 x86 Maschinensprache

CISC: Befehllänge zwischen 1 und 15 Bytes pro Befehl
Befehl hat bis zu 6 Bestandteile: Prefix, Opcode, Displacement, Immediate mit variabler Länge,

Prefix	Opcode	Mod R/M	SIB	Displacement	Immediate
$\leq$ 4 opt. 1b	1,2,3b	1b if req.	1b if req.	0,1,4b	0,1,2,4b

- Prefix:
  - REP/REPE/REPNZ/...
  - Segment (CS/DS/FS/...)
  - Operandengröße (16/32/64 Bits)
  - Adressgröße (16/32/64 Bits)
- Opcode: abhängig von Escape-Sequenz wird Opcode mit anderen Tabelle dekodiert
  - `<primary opcode>`
  - `0x0F <primary opcode>`
  - `0x0F 0x38 <secondary opcode>`
  - `0x0F 0x3A <secondary opcode>`
- Mod R/M: beschreibt Adressierungsart und verwendete Operanden
  - 2-Bit: Mode
    11: kein Speicherzugriff, nur Register
    10: Speicherzugriff per SIB 32-bit Displacement
    01: Speicherzugriff per SIB und 8-bit Displacement
    00: siehe R/M
  - 3-Bit: Register
    000: EAX, 001: ECX, 010: EDX, 011: EBX, 100: ESP, 101: EBP, 110: ESI, 111: EDI
  - 3-Bit: R/M
    falls Mode=11: Register wie oben
    falls Mode=00:
    - 100: Speicherzugriff per SIB ohne Displacement
    - 101: Displacement-only Speicherzugriff
- Scale / Index / Base
  - Scale (2-bit): Faktor 00 = 1, 01 = 2, 10 = 4, 11 = 8
  - Index (3-bit): Register wie zuvor
  - Base (3-bit): Register wie zuvor
- Displacement: bei Offset-Adressierung
- Immediate: für Konstanten

3 Hochsprachen

Basic Blocks: Folge von Befehlen, die in gegebener Reihenfolge ausgeführt werden und folgendes erfüllen:
- ex. genau ein Eingang und genau ein Ausgang (Start/Endbefehl)
- nur letzter Befehl kann Verzweigung sein
- nur erster Befehl darf Ziel einer Verzweigung sein

Kontrollflussgraph: Knoten sind Basic Blocks, Kanten repräsentieren Kontrollfluss dazwischen
Switch-Tabelle: Sprung mittels indirekter Adressierung nach `[RDI*8 + table_offset]` / `[EDI*4 + table_offset]`.
für Fallunterscheidungen kompakter Wertebereiche, sonst z.B. zweistufige Switch-Tabellen oder Binärbaum
Kopf- vs. Fußgesteuerte Schleifen: while/for beginnen mit Sprung zu Vergleich, do-while durchläuft zuerst Block

3.1 Datenstrukturen

- Globale/statische Variablen: Datensegment
- Lokale Variablen: Stack unterhalb EBP
- Parameter: Stack oberhalb EBP

Breite ganzzahliger Datentypen anhand von MOV:
MOVB = char, MOVW = short, MOVL = int/long, MOVQ = long/long long
Vorzeichen lassen sich nicht durch Initialisierung eines Registers/Speicherstelle bestimmen.
⇒ Typkonvertierungen, arithmetische Shifts, bedingte Sprünge
VZ-los: MOVZX, MOVZBW, MOVZBL, MOVZWL, JA, JAE, JB, JBE, JNA, JNAE, JNB, JNBE, SHR
VZ-behaftet: MOVSX, MOVSBW, MOVSBL, MOVSWL, JG, JGE, JL, JLE, JNG, JNGE, JNL, JNLE, SAR, CBW, CWD, CWDE, CDQ

Arrays: wenn über variable Indizes adressiert, leicht von anderen Variablentypen unterscheidbar
Structs: wenn Strukturzeiger übergeben und Elemente in Folge per `->` referenziert: Identifizierung möglich

Division hängt von Signedness ab:
- VZ-los: fülle EDX mit 0x0
- VZ-behaftet: wenn pos. VZ, fülle EDX mit 0x0, sonst mit 0xF..F (`sar edx, 0x1f / 0x3f`)
- Optimierung bei Div durch 2^x mit Shift um x
  bei neg., ungeraden Zahlen muss davor $2^{x+1} - 1$ addiert werden
- Optimierung bei MUL durch Shift + Addition

3.2 GCC Optimierungsstufen

- O0: keine Optimierung → schnelle Kompilation, geeignet zum Debuggen
- O1: Reduzierung von Codegröße und -geschwindigkeit
- O2: O1 plus Optimierungen, die Datei leicht vergrößern (Alignment)
- O3: O2 plus Optimierungen, die Datei stark vergrößern (Inlining, Loop Unswitching)
- Ofast: Stärker als O3, Einhaltung von C Standars wird missachtet
- Os: Größenoptimierung (mögl. kleine Datei)

Optimierungen:
- Propagierung von Konstanten
- Subroutine Inlining
- Loop Unswitching
- Loop Unrolling
- Branchless Code (ersetze Code durch sprungfreien Code)
- Frame Pointer Omission (Referenzierung über SP anstatt BP, weiteres freies Register)

4 Reversing Werkzeuge

4.1 Statische Analyse

+ sicherer bei Analyse von Schadsoftware, alle Programmpfade abgedeckt
+ kein Fehlverhalten durch Anti-Debugging Techniken, Analyse von nicht-ausführbaren Formaten
- unübersichtlich, aufwendig, durch Obfuscation stark beeinträchtigt

Werkzeuge:
- Disassembler: IDA Pro, objdump, ndisasm, Ghidra, Radare2, etc.
  Urproblem bei von-Neumann: Code oder Daten?
  - Linear Sweep: Ende einer Instr. leitet Beginn der nächsten ein
  - Recursive Traversal: Sequentiell wie bei Linear Sweep, Sprünge werden rekursiv verfolgt

- Decompiler: Hex-Rays Decompiler (IDA PRo), Ghidra, etc.
  Komplizierte Sache, tut meist mittelgut, da Compilierung von C verlustbehaftet ist (many-to-many Relation)

4.2 Dynamische Analyse

Blackbox: Beobachtung der I/O-Verhaltens (geöffnete Dateien, Netzwerk, ...)
Whitebox: Beobachtung von Speicher/Registern/Instruktionen, mit Breakpoints, Debugger, ...

+ Programme entpacken sich beim Start
+ I/O-Verhalten beobachtbar
+ Programmpfade (Instantiierungen von Variablen) zur Laufzeit bekannt
+ einfache Erkennung der Hot Parts eines Programms

Empfehlung: Ausführung in VMs

- schützt vor Infektionen mit Schadsoftware
- Ursprungszustand wiederherstellbar
- Hardware kann angepasst werden, redirect auf eigene Server, etc.

Usermode-Debugger: x64dbg, OllyDBG, IDA PRo, GDB, ...
Linux-Debugging mit ptrace (Debugging-API des Linux Kernels):

- Debugger-Prozess erzeugt Kind (fork)
- Kind-Prozess ruft TRACEME auf (ptrace) und startet Debuggee (exec)
- Debugger empfängt alle Signale vom Debuggee: warte auf Signal (wait), behandle es, führe Debuggee fort(ptrace)

TRACEME (initialer Aufruf von Kind), PEEKUSER (lese Daten vom Speicher des Kindes), POKEUSER (schreibe Daten in Speicher des Kindes), GETREGS, GETSIGINFO, CONT, ATTACH, DETACH
Breakpoints:

- Software: überschreibe Adresse X mit 0xCC (int 3), stelle Instruktion wieder her und dekrementiere IP
- Hardware: x86 unterstützt 4 Breakpoint Register in HW (DB0 - DB3, DB6/DB7 de/aktivieren Breakpoints)

5 Betriebssysteme

5.1 x86

Ziel: Isolation von Betriebssystemkern gegenüber Anwendungen, ebenso Anwendungen gegenüber Anwendungen
x86 bietet Privilegierungsringe mit Kernel Mode (0 = höchste) und User Mode (3 = niedrigste)
Damit verbunden: privilegierte Instruktionen - führen außerhalb von Ring 0 zu general protection exception
Syscalls: Ring-Wechsel für privilegierten Code

5.1.1 User/Kernel Space

- 32-Bit Linux: 3GB User, 1GB Kernel Space
- 32-Bit Windows: 2GB User, 2GB Kernel Space
- Virtueller IA-32 Adressraum:

`0xFFFFFFFF`	Kernel
`0xC0000000`	Stack
	(wachsen zueinander)
`0x09......`	Heap
	BSS (glob./stat. Var: nicht-init.)
	Data (glob./stat. Var: init.)
`0x08048000`	Code
`0x00000000`	

- Virtueller AMD64 Adressraum: 48 Bits zur Adressierung genutzt
  kanonische Form: Bit 48 bis 63 müssen Bit 47 entsprechen (= Sign Extension)

`0xffffffffffffffff`	Kernel
`0xffff800000000000`	
	(nicht kanonische Adressen)
`0x00007FFFFFFFFFFF`	Stack
	(wachsen zueinander)
`0x0000000000800000`	Heap
`0x0000000000600000`	Data
`0x0000000000400000`	Code
`0x0000000000000000`	

- seit Anfang 2018: Kernel Page-Table Isolation: jeder Syscall erfordert Art Kontextwechsel

5.1.2 x86 Betriebsarten

- Real Mode (16 bit)
  Segmente: Segmentregister (verschoben um 4) auf Adresse draufrechnen (ergibt 20bit Adressen), können sich überlappen
- Protected Mode (32 bit)
  Protected Address Mode: `logisch -(Segmentation)-> linear -(Paging)-> physisch`
  16-Bit Segment Register zeigen auf Deskriptor in Deskriptor Tabelle
  Paging: Page, 4K oder 4MB groß, jede Page bekommt physischen Frame zugeordnet
  MMU zentraler Baustein: Page Directory (an CR3, Page Directory Base Register, wechselt bei Context Switch) + Page Table Entry + Offset
  Neben Basisadresse noch weitere Informationen in Page Table Entries gespeichert
- Long Mode (64 bit)
  Paging wie bei Protected Mode, aber größer und mit vierstufigen Pagetables
  Neuerung: NX-Bit, pro Page kontrollierbar, ob ausführbar (davor nur pro Segment)
  CS/DS/ES/SS werden ignoriert, FS/GS anderweitig verwendet

5.2 Windows

Anwendungen sollen keine Systemaufrufe selbst tätigen, sondern die Win32 API nutzen

- `KERNEL32.DLL`: Dateisystem, Speicherverwaltung, Prozess/Threadverwaltung
- `GDI32.DLL`: Pixel/Linien zeichnen, Bitmaps darstellen, ...
- `USER32.DLL`: Windows-style GUI, Fenster, Menüs, ...
- `COM*.DLL`: common controls, Dialoge für Statusbars, Tabs, ...
- `NTDLL.DLL`: weitestgehend undokumentierte Funktionen, können sich ändern

Windows NT Bootprozess:
BOOTMGR, NTLDR / winload.exe → Kernel (NTOSKRNL, HAL.DLL, Treiber) → Session Manager → Graphischer Modus, Logon Manager → LSASS (Local Security Authority Subsystem Service), SCM (Service Control Manager)
Threads: vom OS eigener Stack, Heap sowie Daten- und Code-Sektionen gemeinsam genutzt
Fiber: User-Level Thread

Prozess: PID, mind. 1 Thread, Liste offener Handler, Access Token, ...
Thread: zusätzl. TID, CPU Register Kontext, Stack, Thread Local Storage, Exception Handler Liste, ...

PEB (`fs:[0x30]`, Process...) und TEB (`fs:[0x18]`, Thread Env. Block), TEBs verweisen auf übergeord. PEB

5.3 Linker und Loader

Objektdateien: nicht-ausführbare Binärdateien, relocatable Maschinencode, Symbole, ...
Symbol Stripping: entfernt unnötige Symbole aus finaler Binärdatei → schwerer zu analysieren

- statisches Linken: Auflösung von Bibliotheksabhängigkeiten einmalig zur Linkzeit
  → schneller Programmstart, einfacher zu verteilen
- dynamisches Linken: Auflösung von Bibliotheksabhängigkeiten zur Laufzeit
  → weniger Speicherverbrauch, Bibliothekupdates einfach möglich

5.3.1 Windows DLLs

selbes Format wie EXE (Portable Executable) Code Sections nur einmal im Speicher, Data sections pro Prozess
Relocation/Rebasing: Verschiebung von Code nötig, falls DLLs diesselben virtuellen Adressen beanspruchen

- Relocatable Code: Code muss vom Lader beim Verschieben überarbeitet werden (Windows DLLs)
- Position Independent Code: Code kann von jeder Adresse aus ausgeführt werden (Linux Shared Libraries)

Bibliotheksfunktionen: via Funktionsnamen, Ordinalzahl oder Codeadresse (RVA = Addr. relativ zur Basisaddr.)
Win API Namensschema: `<Präfix><Operation><Objekt><Suffix>`
Präfixe:

- Dbg: Debug
- Ex: Executive Support Routines
- FsRtl: File System Runtime Library
- Hal: Hardware Abstraction Layer
- Io
- Ke: Kernel
- Mm: Memory Manager
- Nt: Windows system services
- Ps: Process Support
- Rtl: Run-time Library
- Zw: Spiegel für Nt-Funktionsaufrufe aus Kernel Mode heraus

Suffixe:

- A: ANSI Format
- W: Unicode Format

Spaß mit DLLs für den Reverser:

- Run-Time Dynamic Linking: Nachladen von DLLs zur Laufzeit, Nutzung z.B. für Import Hiding
- DLL Hijacking: Bibliotheken, die im Programmordner der Anwendung abgelegt sind, höhere Prio als systemweit verfügbare Bibliotheken
- DLL Preloading Angriff: Bibliotheken, ohne kompletten Pfad eingebunden, können überladen werden
- DLL Injection: injiziere fremde, nicht vom Prozess selbst geladene DLL in Prozessadressraum
- DLL Hooking: Library-Calls analysieren bzw. modifizieren
  erste Instr. einer Win32 API Funktion ist `mov edi,edi`, davor 5 Bytes Patch Space

5.3.2 PE Format

Einheitliches Format für 32- und 64-Bit Windows

- DOS-Header: bis heute abwärtskompatibel - magic bytes "MZ"
- PE-Header:
  - magic bytes "PE"
  - File Header (CPU-Typ, Anzahl Sections, Zeitstempel, ...)
  - Optional Header (Linkerversionen, Größe von Code/Daten, Entrypoint, ...)
  - Data Directory (Verweise auf versch. Tabellen, z.B. Export, Import, Resource, IAT, ...)
- Section Table: Tabelle mit vielen Headern
  enthalten Namen, VirtualSize, VirtualAdress, PointerToRelocations, Characteristics, ...
- Section 1, ..., n

6 Softwareschutz

6.1 Hardware-basierter Softwareschutz

Client-Server Modell: Ausführung von kritischem Code auf Server oder Dongle
Trusted Computing: TPM, Intel TXT, Trusted Execution Environment: Intel SGX, AMD SEV, ARM TrustZone

Software Guard Extensions (SGX):
Erweiterung des klassischen Ring-/Isolationskonzepts: isoliere Anwendungen gegenüber Betriebssystem
SGX erlaubt es Anwendungen in Adressraum sog. Enklaven zu erstellen, die sicheren Container darstellen
Jeder Zugriff von außen verboten ⇒ neuer CPU-Modus: Enclave Mode - außerhalb des trad. Ring-Konzepts
MEE (Memory Encryption Engine): RAM nicht vertrauenswürdig → Pages bei Auslagerung verschlüsseln

6.2 Software-basierter Softwareschutz

6.2.1 Schutz vor statischer Analyse: Code Obfuscation

Obfuskierer O ist Programmtransformation auf Eingabe p, sodass gilt:

- Funktionalität: $O(p)$ berechnet dasselbe wie p
- Effizienz: $O(p)$ ist nicht exponentiell langsamer als p
- Verschleierung: $O(p)$ ist unverständlicher als p
- Virtual Black Box Obf.: alle Informationen, die effizient aus $O(p)$ bestimmt werden können, können auch durch reinen Black-Box Zugriff auf p bestimmt werden
- Best Possible Obf.: $O(p)$ verrät weniger Informationen über p als jedes andere Programm mit derselben Funktionalität wie p
- Time Limited Black Box Obf.: Virtual Black Box Eigenschaft gilt mind. für Zeitraum t
- Indistinguishable Obf.: p, p' zwei unterscheidbare Programme, $O(p)$ und $O(p')$ nicht unterscheidbar

In der Praxis: Ziel ist Erschwerung der Analyse, um Zeit zu gewinnen
Obfuskierungstechniken:

- Source-level
- Intermediate-Representation-level
- Binary-level

z.B.

- Compileroptimierungen
- Zeichenketten-Obfuskierung
- Code Blow-Up: Junk/Dead Code Insertion
- Opaque Predicates: verschleierte boolsche Bedingung, deren Auswertung immer denselben Wert liefert
- Fake Loops
- Bogus Control Flow
- Function Merging: falsche Modularisierung, indem zusammenhangslose Funktionen verschmolzen werden
- Function Splitting: falsche Modularisierung durch Funktionenaufteilung
- Arithmetik Obfuskierung: durch teil-homomorphe Verfahren
- Aliasing: Kette von Dereferenzierungen oder versch. Referenzen auf selben Wert
- One Point Functions: Vergleiche zw. Variablen und Konstanten durch Hash-Funktionen verschleiert
- Control Flow Flattening
- Interpreter/VM: zuf. Bytesprache + VM, die diese interpretiert + Programm in diese Bytesprache übersetzen und mit VM ausliefern
- Verschlüsselter Code: konstanter Decrypter-Stub bleibt verräterisch
- Polymorpher Code: ändere Programm mit jeder Replikation nach Außen hin vollständig
  - Decrypter-Stub wird von Mutation Engine neu generiert
  - Payload wird mit wechselnden Keys und Algorithmen verschlüsselt
- Metamorpher Code: Metamorphie-Engine transformiert das gesamte Programm (inkl. sich selbst)

6.2.2 Windows-spezifische Obfuskierung

- Verwendete Bibliotheken sagen viel über Programmverhalten aus (als Heuristik in AV-Software)
  Import Hiding: kernel32!LoadLibrary(), kernel32!GetProcAddress()
  Problem: Klartextstrings ⇒ Hash(name) + iteriere alle Funktionen der Bibliothek → passender Eintrag
- Structured Exception Handling: absichtliche Exceptions, regulärer Code im Exception Handler
  statische Analyse erschwert durch verschleierten CFG
  dynamische Analyse erschwert durch Übergabe der Exceptions an den Debugger

6.2.3 Windows/PE Packer

- ursprünglich für Kompression ausführbarer Dateien
- Entpacker ist Teil der ausführbaren Datei, dekomprimiert sich zur Laufzeit
- verschleiert Payload → keine Disassemblierung, verschleierte Zeichenketten, ...

UPX: Ultimate Packer for Executables
UPX0 und UPX1 Sections - in UPX1 zuerst Entpackroutine, danach Sprung an Original Entry Point in UPX0
Unpacking:

- OEP ausfindig machen, Breakpoint setzen
- Programm sich selbst entpacken lassen
- Speicherabbild des entpackten codes ziehen
- Einstiegspunkt und Import Tabelle korrigieren

Code Virtualizer: erzeuge zuf. Bytesprache, erzeuge passende VM, liefere diese mit übersetztem Programm aus

6.2.4 Schutz vor dynamischer Analyse

- Anti-Debugging:
  - IsDebuggerPresent()
  - CheckRemoteDebuggerPresent()
  - BeingDebugged-Flag im PEB gesetzt?
  - Trace-Flag im EFLAGS gesetzt?
  - Prozesstabellen nach Debuggern durchsuchen
  - Fenster mit Namen "OllyDbg" offen? (FindWindow())
  - Dateisystem- oder Registry-Einträge von Debuggern
  - Zeitmessung (per GetTickCount() oder rdtsc)
  - Breakpoint Register DR0 bis DR4 selbst belegen
  - 0xCC (INT 3) in den Kontrollfluss einbauen
- Anti-Emulation: AV-Software emuliert unbekannte Programme, dass sich Malware entpackt - Viren versuchen durch Zeitverzögerung, das zu umgehen
- Anti-Dumping: Löschen/Überschreiben des PE Headers, Manipulation der IAT, Seitenweise Ent- bzw. Verschlüsselung
- Anti-Virtualization: Erkennung virtueller Maschinen, z.B. anhand von Hardware-Eigenschaften oder Timing, Location der IDT, ...
- Tamperproofing: Erkennung und Verhinderung von Manipulationen, z.B. durch CRC, Hashfunktion, ... liegen selbst im Hauptspeicher → Cross-Checking

7 Exploitaiton

7.1 Stack-basierte Buffer Overflows

Auf Stack liegen Kontrollinformationen zw. Parametern, lokalen Variablen und Puffern.
→ gezielte Manipulation ermöglicht Steuerung des Kontrollflusses
Overflow: Kontrolliere alte Stack-Frames, Underflow: überschreibe neuere Stack Frames
Pufferüberläufe in C: keine Überprüfung, unsichere Bibliotheksfunktionen → Shellcode Injection
Shellcode: kein eigenes Datensegment, keine Nullbytes
Unter Windows schwieriger, da kein `int 0x80`: Basisadresse der kernel32.dll, GetProcAddress, WinExec, ...

7.2 Integer Errors

Überlauf von Wertebereichen, z.B. durch Typkonvertierungen
Vorzeichenfehler: Programmierer nutzen int, C Bibliotheksfunktionen erwarten unsigned
Off-by-One Fehler, kann zu Frame Pointer Overwrite führen

7.3 Heap Schwachstellen

Heap Overflow:
Verwaltung des Heaps im User Mode implementiert, Speicherblöcke in Form sog. Chunks verwaltet
Freie Heap Chunks werden in Form einer doppelt verketteten Liste verwaltet
Liste wird nach Größe sortiert, benutzte Blöcke aus Liste ausgetragen, FWD- und BCK-Zeiger frei für Daten
Idee: FWD- und BCK-Zeiger manipulieren, dass beim nächsten free bel. Speicher überschreiben (z.B. RIP)
Kommt es durch free zur Vereinigung zweier aufeinander folgender Chunks, wird unlink Macro aufgerufen:
`chunk->fwd->bck = chunk->bck`, `chunk->bck->fwd = chunk->fwd`
Dadurch steht Blödsinn in den anderen Pointern.

7.3.1 Double Free

Undefiniertes Verhalten - kann passieren, dass derselbe Chunk doppelt in Liste freier Chunks gespeichert ist
→ zwei Sichtweisen auf ein- und denselben Speicherbereich
→ ersten 8 Bytes des allokierten Speichers sind Kontrollinformationen des freien Chunks
→ Manipulation von FWD und BCK - dann Ausnutzung des UNLINK Makros

7.4 Formatstring Schwachstelle

Software uses externally-controlled format strings in printf-style functions
Stack-Inhalte auslesen, Programmabsturz, oder sogar Programmfluss manipulieren:

- mittels `%n` kann in den Speicher geschrieben werden: Anzahl der bisher geschriebenen Chars
- mittels `%mX` lässt sich Anzahl ausgegebener Zeichen genau steuern
- One-Shot Methode: wenn Formatstring selbst auf dem Stack: überschreibe RIP mit Zeiger auf Benutzereingabe
- Short-Write Methode: Problem der One-Shot Methode: zeitaufwendig, um reale Adressen zu schreiben
  → Schreiboperation in zwei Operationen aufteilen (`%hn`, `%hhn`)

7.5 NULL Pointer Dereferenzierung

User Mode Exploitation: Angreifer kann potentiell an beliebige Speicherstellen schreiben
Kernel Mode Exploitation: Bsp.: uninit. func-ptr. NULL liegt im User Space → mappe Shellcode an 0x0

7.6 Address Space Layout Randomization (ASLR)

Segmente im virtuellen Adressraum eines Prozesses werden zufällig angeordnet
erschwert Anspringen von Shellcode, keine fixen Adressen mehr

- Stack Adressen in Erfahrung bringen: `/proc/<pid>/stat`
- Brute Force / Heap Spraying / NOP Slides
- Return-oriented Programming

Mehr Techniken:

- ret2ret:
  Idee: Pointer, die auf Stack-Variable zeigen, wissen korrekte Stack-Adressen
  Angriffsidee: nutze Pointer auf Stack-Variablen als Sprungbrett zu Shellcode
  Problem: Pointer muss an Stelle des RIP kopiert werden → Kette von return-Befehlen
  NOP-Slide: Pointer zeigt "ungefähr" auf Code - Slide zu dem Rest
- ret2pop:
  Durch Parameterübergabe liegt oft ein 'perfekter Zeiger' auf dem Stack, der schon auf den Buffer zeigt
  ret to ret to ret to ... to pop-ret
- jmp2esp:
  Während des Funktionsepilogs ist Ziel des ESP eine vorhersagbare Speicherstelle auf dem Stack
  Angriffsidee: überschreibe RIP mit Zeiger auf Instruktion 'jmp esp'. Interpretiere hierzu Code als Daten.
- call eax:
  Stringoperationen geben über EAX Zeiger auf Zielstring zurück.
  Angriffsidee: platziere Shellcode am Anfang des Strings, überschreibe RIP mit Zeiger auf call eax.

7.7 Data Execution Prevention (DEP)

z.B. mit NX-Bit: pro Page kontrollieren, ob diese ausführbar ist
verhindert nur die Ausführung von Shellcode auf dem Stack, nicht die Manipulation des RIP
Angriffe:

- return-into-libc: führe existierende Funktionen im Codesegment aus (z.B. um Page auf executable zu setzen)
- return-oriented programming: Wiederverwendung von vorhandenen Codefragmenten

7.8 Compiler-level Protections

Stack Canaries: zuf. Wert auf Stack, dessen Integrität beim Funktionsepilog geprüft wird
Stack Shield: auf Shadow Stack werden redundante Kopien der RIPs gespeichert