Software Reverse Engineering (SRE)

SRE

- **Software Reverse Engineering**
  - Also known as Reverse Code Engineering (RCE)
  - Or simply “reversing”
- Can be used for **good**...
  - Understand malware
  - Understand legacy code
- ...or **not-so-good**
  - Remove usage restrictions from software
  - Find and exploit flaws in software
  - Cheat at games, etc.

SRE

- We assume…
  - Reverse engineer is an attacker
  - Attacker only has exe (no source code)
  - No bytecode (i.e., not Java, .Net, etc.)
- Attacker might want to
  - Understand the software
  - Modify (“patch”) the software
- SRE usually focused on Windows
  - So we focus on Windows

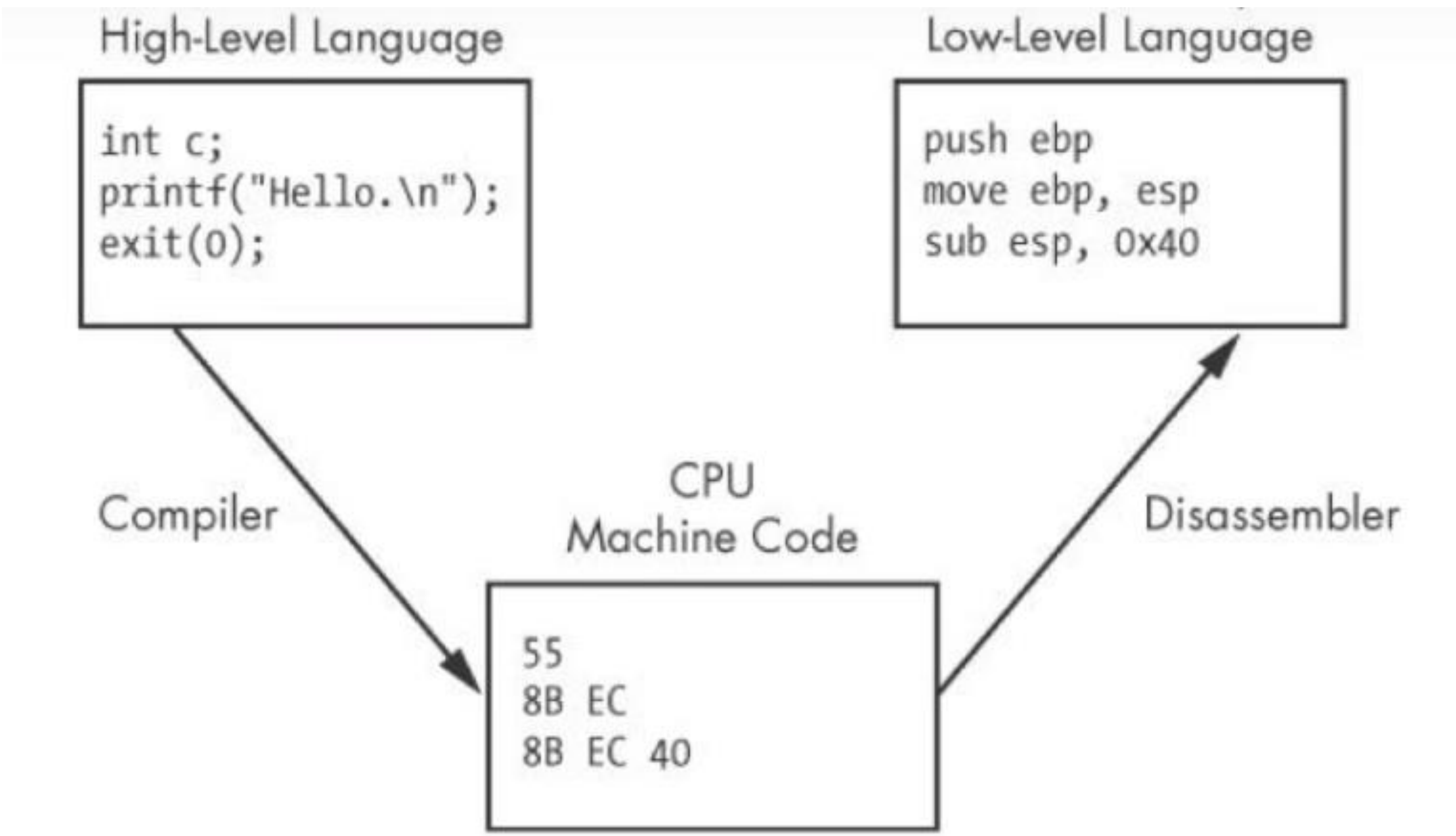
SRE Tools

- Disassembler
  - Converts exe to assembly (as best it can)
  - Cannot always disassemble 100% correctly
  - In general, not possible to re-assemble disassembly into working executable
- Debugger
  - Must step thru code to completely understand it
  - Labor intensive — lack of useful tools
- Hex Editor
  - To **patch** (modify) exe file
- VMware

SRE Tools

- **IDA Pro** — good disassembler/debugger
  - Costs a few hundred dollars (free version exists)
  - Converts binary to assembly (as best it can)
- **OllyDbg** — high-quality shareware debugger
  - Includes a good disassembler
- **Hex editor** — to view/modify bits of exe
  - UltraEdit is good — freeware
  - HIEW — useful for patching exe

Disassembly Process



Why is Debugger Needed?

- Disassembly gives **static** results
  - Good overview of program logic
  - User must “mentally execute” program
  - Difficult to jump to specific place in the code
- Debugging is **dynamic**
  - Can set break points
  - Can treat complex code as “black box”
  - And code not always disassembled correctly
- Disassembly ***and*** debugging ***both*** required for any serious SRE task

SRE Necessary Skills

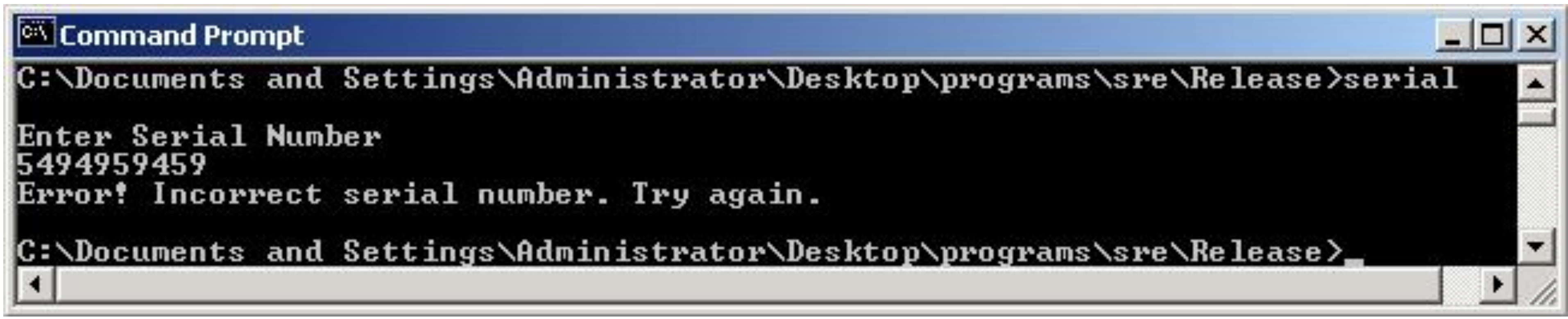
- Working knowledge of target assembly code
- Experience with the tools
  - IDA Pro — sophisticated and complex
  - **OllyDbg** — good choice for this class
- Knowledge of Windows **Portable Executable** (PE) file format
- Boundless patience and optimism
- SRE is a tedious, labor-intensive process!

SRE Example

- We consider a simple example
- This example only requires disassembly (IDA Pro used here) and hex editor
  - Trudy disassembles to understand code
  - Trudy also wants to patch (modify) the code
- For most real-world code, would also need a debugger (e.g., OllyDbg)

SRE Example

- Program requires serial number
- But Trudy doesn't know the serial number…



- Can Trudy get serial number from exe?

SRE Example

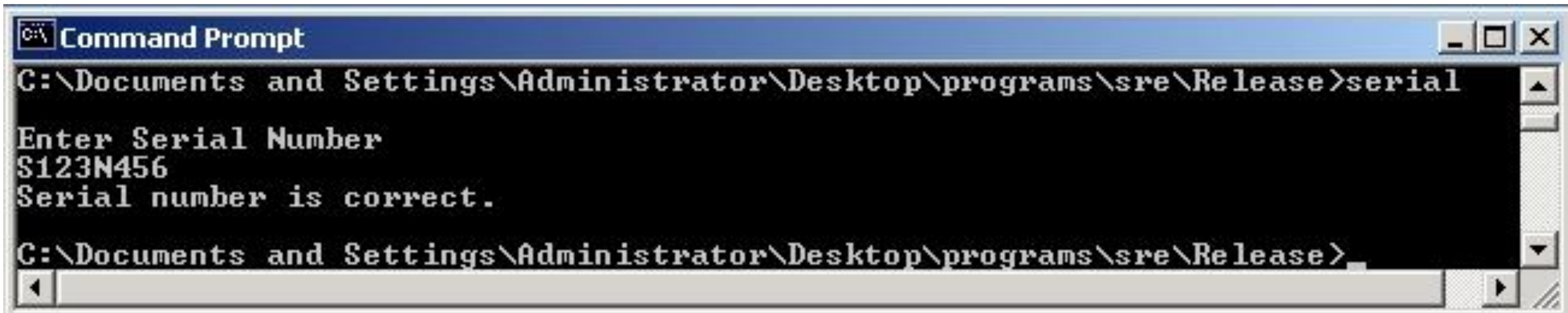
- IDA Pro disassembly

```
.text:00401003        push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008        call    sub_4010AF
.text:0040100D        lea     eax, [esp+18h+var_14]
.text:00401011        push    eax
.text:00401012        push    offset aS       ; "%s"
.text:00401017        call    sub_401098
.text:0040101C        push    8
.text:0040101E        lea     ecx, [esp+24h+var_14]
.text:00401022        push    offset aS123n456 ; "S123N456"
.text:00401027        push    ecx
.text:00401028        call    sub_401060
.text:0040102D        add     esp, 18h
.text:00401030        test    eax, eax
.text:00401032        jz      short loc_401045
.text:00401034        push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039        call    sub_4010AF
```

- Looks like serial number is S123N456

SRE Example

- Try the serial number S123N456



- It works!
- Can Trudy do "better"?

SRE Example

- Again, IDA Pro disassembly

```
.text:00401003        push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008        call    sub_4010AF
.text:0040100D        lea     eax, [esp+18h+var_14]
.text:00401011        push    eax
.text:00401012        push    offset aS        ; "%s"
.text:00401017        call    sub_401098
.text:0040101C        push    8
.text:0040101E        lea     ecx, [esp+24h+var_14]
.text:00401022        push    offset aS123n456 ; "S123N456"
.text:00401027        push    ecx
.text:00401028        call    sub_401060
.text:0040102D        add     esp, 18h
.text:00401030        test    eax, eax
.text:00401032        jz      short loc_401045
.text:00401034        push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039        call    sub_4010AF
```

- And hex view...

```
.text:00401010  04 50 68 84 80 40 00 E8-7C 00 00 00 6A 08 8D 4C
.text:00401020  24 10 68 78 80 40 00 51-E8 33 00 00 00 83 C4 18
.text:00401030  85 C0 74 11 68 4C 80 40-00 E8 71 00 00 00 83 C4
.text:00401040  04 83 C4 14 C3 68 30 80-40 00 E8 60 00 00 00 83
```

SRE Example

```
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_4010AF
.text:0040100D                 lea     eax, [esp+18h+var_14]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401098
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+24h+var_14]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401060
.text:0040102D                 add     esp, 18h
.text:00401030                 test    eax, eax
.text:00401032                 jz      short loc_401045
.text:00401034                 push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039                 call    sub_4010AF
```

- "test eax,eax" is AND of eax with itself
    - So, zero flag set only if eax is 0
    - If test yields 0, then jz is true
- Trudy wants jz to always be true
- Can Trudy patch exe so jz always holds?

SRE Example

- Can Trudy patch exe so that jz always true?

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS       ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    xor     eax, eax
.text:00401032    jz      short loc_401045     ← jz always true!!!
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

Assembly		Hex
test	eax,eax	85 C0 …
xor	eax,eax	33 C0 …

SRE Example

- Can edit serial.exe with hex editor

serial.exe

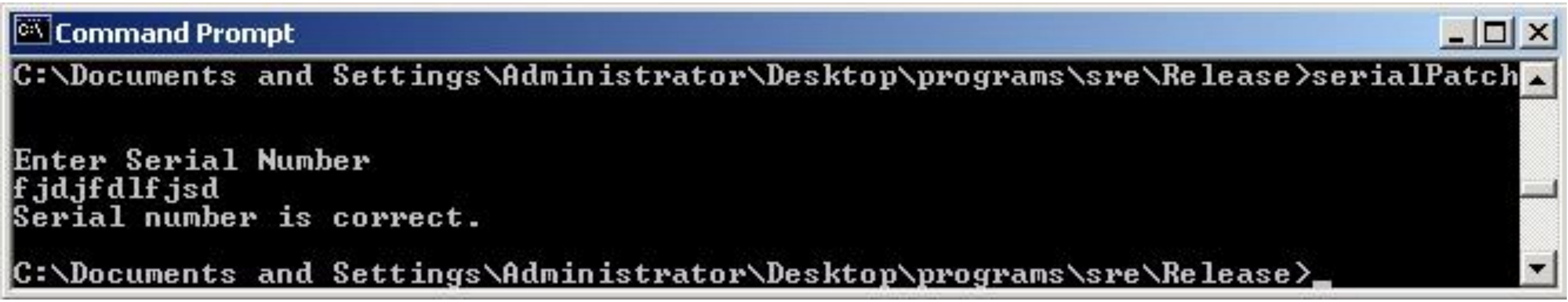
```
00001010h: 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C
00001020h: 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18
00001030h: 85 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4
00001040h: 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83
00001050h: C4 04 83 C4 14 C3 90 90 90 90 90 90 90 90 90 90
```

serialPatch.exe

```
00001010h: 04 50 68 84 80 40 00 E8 7C 00 00 00 6A 08 8D 4C
00001020h: 24 10 68 78 80 40 00 51 E8 33 00 00 00 83 C4 18
00001030h: 33 C0 74 11 68 4C 80 40 00 E8 71 00 00 00 83 C4
00001040h: 04 83 C4 14 C3 68 30 80 40 00 E8 60 00 00 00 83
00001050h: C4 04 83 C4 14 C3 90 90 90 90 90 90 90 90 90 90
```

- Save as serialPatch.exe

SRE Example



- **Any** “serial number” now works!
- Very convenient for Trudy

SRE Example

- Back to IDA Pro disassembly…

serial.exe

```
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_4010AF
.text:0040100D                 lea     eax, [esp+18h+var_14]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401098
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+24h+var_14]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401060
.text:0040102D                 add     esp, 18h
.text:00401030                 test    eax, eax
.text:00401032                 jz      short loc_401045
.text:00401034                 push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039                 call    sub_4010AF
```

serialPatch.exe

```
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_4010AF
.text:0040100D                 lea     eax, [esp+18h+var_14]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401098
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+24h+var_14]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401060
.text:0040102D                 add     esp, 18h
.text:00401030                 xor     eax, eax
.text:00401032                 jz      short loc_401045
.text:00401034                 push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039                 call    sub_4010AF
```

SRE Attack Mitigation

- **Impossible** to prevent SRE on open system
- Can we make such attacks more difficult?
- Anti-disassembly techniques
  - To confuse static view of code
- Anti-debugging techniques
  - To confuse dynamic view of code
- Tamper-resistance
  - Code checks itself to detect tampering
- Code obfuscation
  - Make code more difficult to understand

Anti-disassembly

- Anti-disassembly methods include
  - Encrypted or “packed” object code
  - False disassembly
  - Self-modifying code
  - Many other techniques
- Encryption **prevents** disassembly
  - But need plaintext decryptor to decrypt code!
  - Same problem as with polymorphic viruses

Anti-disassembly Example

- Suppose actual code instructions are

inst 1	jmp	junk	inst 3	inst 4	…

- What a "dumb" disassembler sees

inst 1	inst 2	**inst 3**	**inst 4**	**inst 5**	**inst 6**	…

- This is example of "false disassembly"
- Persistent attacker will figure it out

Anti-debugging

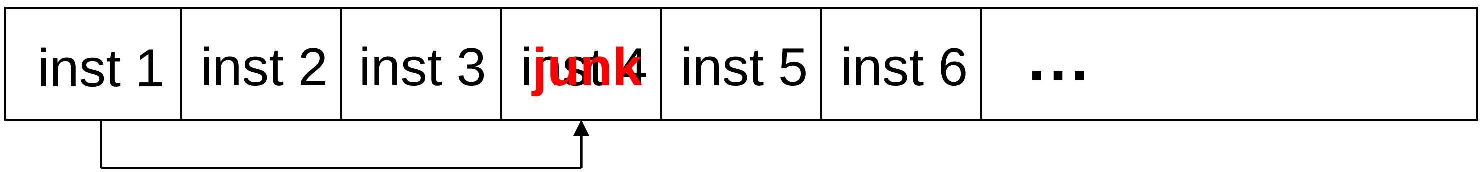
- Can also monitor for
  - Use of debug registers
  - Inserted breakpoints
- Debuggers don't handle ***threads*** well
  - Interacting threads may confuse debugger…
  - …and therefore, confuse attacker
- Many other debugger-unfriendly tricks
  - See next slide for one example

Anti-debugger Example

inst 1	inst 2	inst 3	inst 4	inst 5	inst 6	…

- Suppose when **program** gets inst 1, it pre-fetches inst 2, inst 3, and inst 4
  - This is done to increase efficiency
- Suppose when **debugger** executes inst 1, it does ***not*** pre-fetch instructions
- Can we use this difference to confuse the debugger?

Anti-debugger Example



- Suppose inst 1 **overwrites** inst 4 in memory
- Then program (without debugger) will be OK since it fetched inst 4 at same time as inst 1
- Debugger will be confused when it reaches **junk** where inst 4 is supposed to be
- Again, clever attacker can figure this out

Tamper-resistance

- Goal is to make patching more difficult and detectable
- Code can **hash** parts of itself
- If tampering occurs, hash check fails
- Research has shown, can get good coverage of code with small performance penalty
- But don't want all checks to look similar
  - Or else easy for attacker to remove checks
- This approach sometimes called "guards"

Code Obfuscation

- Goal is to make code hard to understand
  - Opposite of good software engineering
  - Spaghetti code is a good example
- Much research into more robust obfuscation
  - Example: **opaque predicate**

    int x,y

      :

    if((x–y)∗(x–y) > (x∗x–2∗x∗y+y∗y)){…}
  - The if() conditional is always false
- Attacker wastes time analyzing dead code

Legality

- *Depends on many factors*
- Always seek legal counsel before getting yourself into any high-risk reversing project
- *Example: Sega v. Accolade*
  - *Accolade violated copyright law and sued by Sega in 1991*

Code Obfuscation

- Code obfuscation sometimes promoted as a powerful security technique
- It has been shown that obfuscation probably cannot provide strong, crypto-like security
- Obfuscation might still have practical uses
  - Even if it can never be as strong as crypto

Authentication Example

- Software used to determine authentication
- Ultimately, authentication is 1-bit decision
  - Regardless of method used (pwd, biometric, …)
  - Somewhere in authentication software, a single bit determines success/failure
- If Trudy can find this bit, she can force authentication to always succeed
- Obfuscation makes it more difficult for attacker to find this all-important bit

Obfuscation

- Obfuscation forces attacker to analyze larger amounts of code
- Method could be combined with
  - Anti-disassembly techniques
  - Anti-debugging techniques
  - Code tamper-checking
- All of these increase work/pain for attacker
- But a persistent attacker may ultimately win

Types of Obfuscation

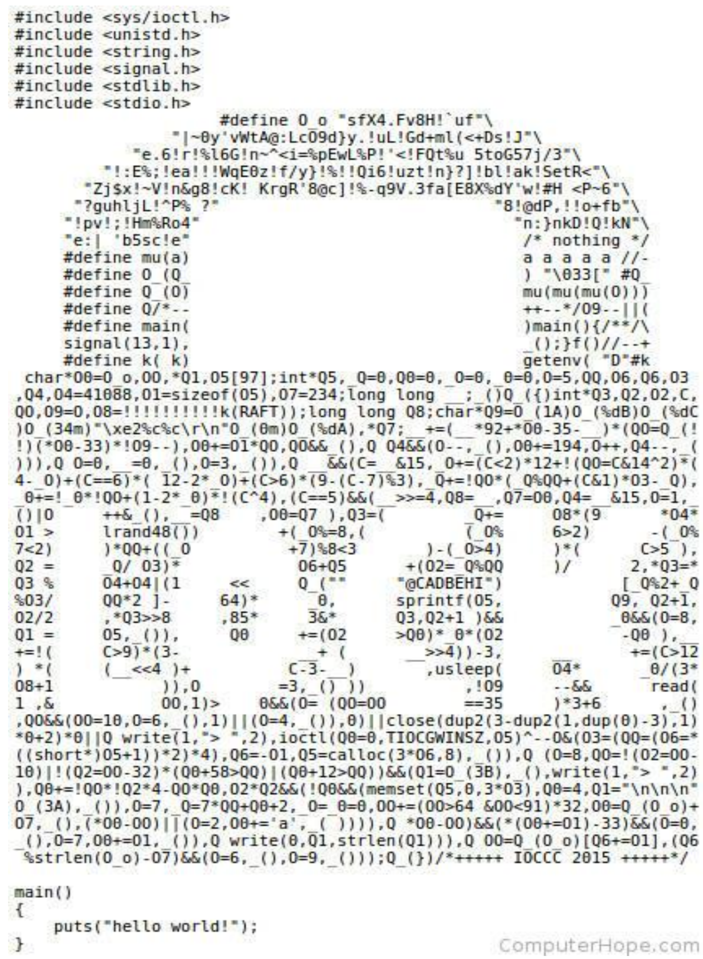
- Obfuscation can be applied depending on the format in which software will be distributed
- Different types-
  - Source Code Obfuscation
  - Java Bytecode Obfuscation
  - Binary Obfuscation

Source Code Obfuscation

- Target is to make source code less intelligible
  - However, the resulting source code should still compile and result in a functionally equivalent program
- Writing bad way of software is a way of making complicated code
  - But, such practice is not commonly used
- It is best added automatically

Source Code Obfuscation

- Manual obfuscation
- International Obfuscated C Code Contest
- www.ioccc.org



Source Code Obfuscation

- Commonly used techniques-
  - Replacing symbol names with non-meaningful one
  - Substitute the constant values with arithmetic expression
  - Removing source code formatting
  - Exploiting the preprocessor

Binary Obfuscation

- Aim: Making the binary representation of software more difficult to understand
- Binary code is a low level code
- So obfuscated binary code will be more difficult for attacker
- Binary obfuscation is achieved through binary rewriting

Binary Rewriting

- Use of exact address
- Use of assembly instruction
- Typically, performed on full program
- Generally, applied as the last step of software development life cycle

Binary Rewriting

- Few limitations-
  - Complicated as many high level information not available in binary code
  - Sometimes easily detectable
    - Code added after register assignment often requires to free registers to perform computation
  - Architecture dependence

Java Bytecode Obfuscation

- Similar to binary obfuscation
- The binary representation of Java Virtual Machine is obfuscated
- Also, it includes virtually all source code information
- Very susceptible to reverse engineering
- Many restrictions are applicable on Java Bytecode Obfuscation
  - But these are not applicable on binary obfuscation

Source Code Transformations for Binary Obfuscation

- Source code obfuscation can also impact on binary
- Let’s consider the following three different classes
  - Layout obfuscation
  - Control flow obfuscation
  - Data obfuscation

Layout Obfuscation

- Exploit the preprocessor to make the code unreadable
- Scramble the identifier
- Change formatting
- Remove comments

Original code

```
int my_output()
{
  int count;
  for (count = 0; count < MAX_INDEX; ++count)
  printf("Hello %d!\n", count);
}
```

Obfuscated code

```
#define a int
#define b printf
#define c for
a l47(){a l118;c(l118=0;l118<0x664+196-
0x71e;++l118)
b("\x48\x65\x6c\x6c\x6f\x20\x25\x64\x21\n",
l118);}
```

However, these layout obfuscation transformations do not survive the compilation phase

Control Flow Obfuscation

- Apply transformation to hide the control flow of a program
  - Opaque predicates
  - Control flow flattening

Opaque Predicates

- Tautological if statement
- True opaque predicates will always evaluate to true
- False opaque predicates will always evaluate to false

```
int a=2,b=3,c=4,d=5;
If((a+b+c*d)>10)
{
   puts("Yes");
   exit(0);
}
puts("No");
```

What compiler does?

```
int a=2,b=3,c=4,d=5;
If((a+b+c*d)>10)
{
   puts("Yes");
   exit(0);
}
puts("No");
```

```
PUSH "Yes"
CALL $PUTS
PUSH 0
CALL $EXIT
```

As the variables are statically defined so compiler can optimize it easily

What compiler does?

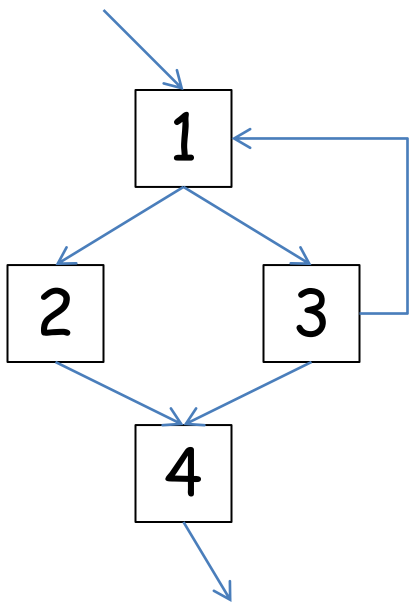
```
int a,b,c,d;
srand(time(0));
a=rand()+1;b=rand()+2;
c=rand()+3;d=rand()+4;
If((a+b+c*d)>0)
{
   puts(“Yes”);
   exit(0);
}
puts(“No”);
```

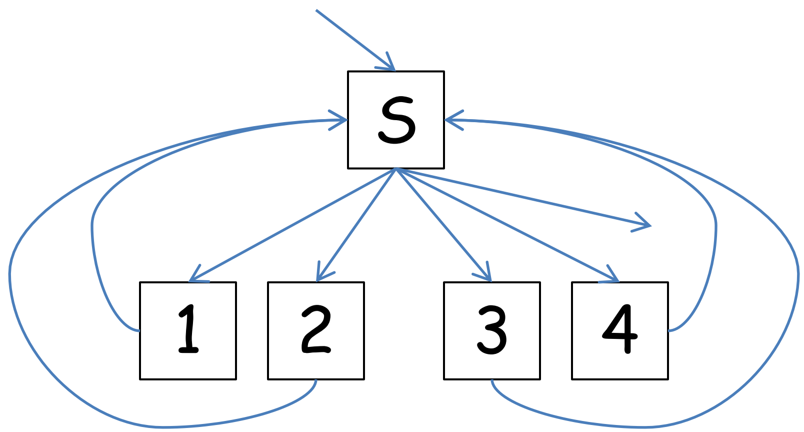
```
TEST EAX,EAX
JLE SHORT :NO
PUSH “Yes”
CALL $PUTS
PUSH 0
CALL $EXIT
NO: PUSH “No”
CALL $PUTS
```

As the values are received dynamically the compiler may not able to optimize it

Control Flow Flattening

- Obscure the control flow of a program
- Tries to flatten the control flow graph

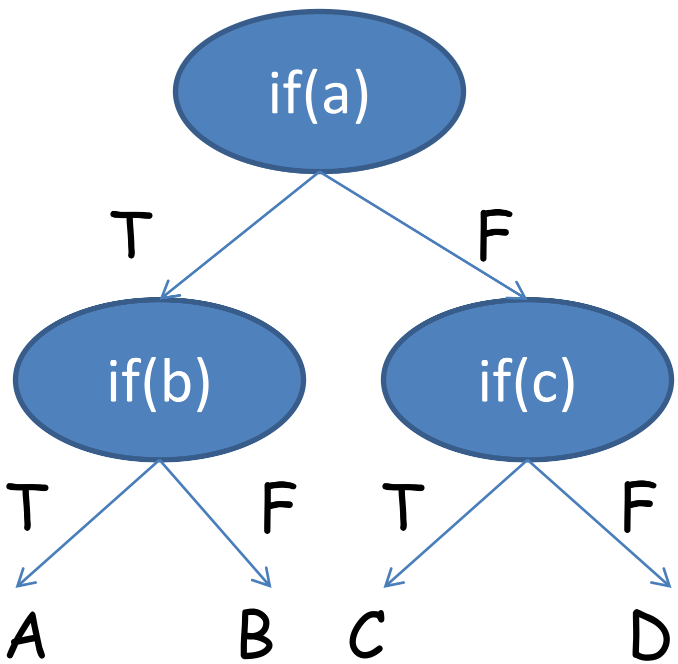


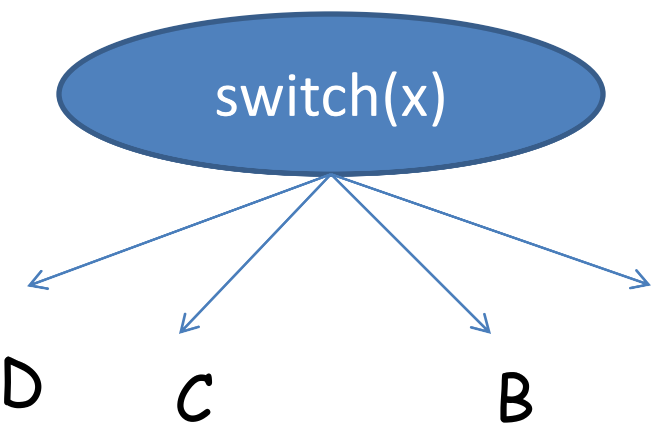


Example

&&: logical AND assumes both the operands as Boolean types
|: bitwise OR can be applied on integral values
<<: left shift operator

`x=(a&&1)<<2|(b&&1)<<1|(c&&1)`

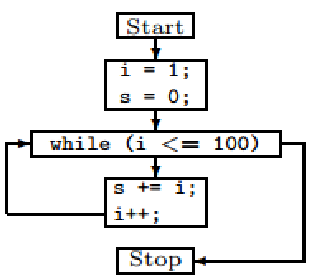




Control Flow Flattening

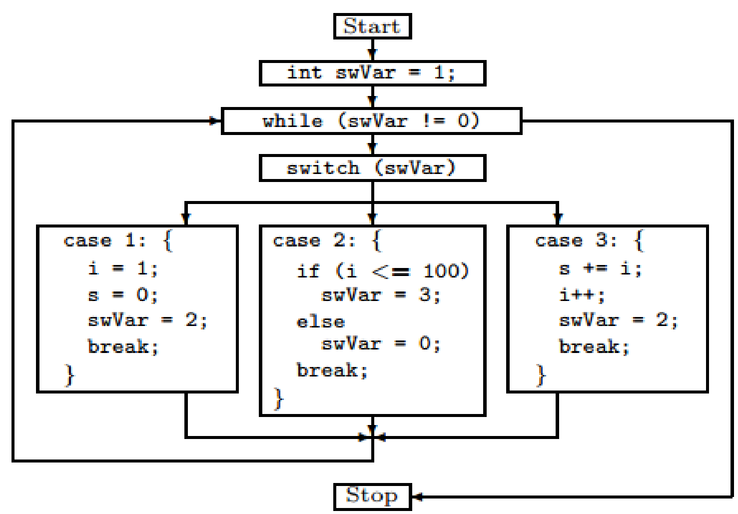
original

```
i = 1;
s = 0;


while (i <= 100) {



  s += i;
  i++;

}
```



control-flow flattening applied

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1: {
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2: {
      if (i <= 100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3: {
      s += i;
      i++;
      swVar = 2;
      break;
    }
  }
}
```



Data Obfuscation

- Data is obfuscated before compilation phase and de-obfuscated during run time
- Requires some more care than control flow obfuscation
- Strings generally don’t lead to what program does, but it can help in reverse engineering

Data Obfuscation

- int x;
- x=7;
- x<<=2;
- x*=2;
- x-=24;
- x<<=1;

x=64

Data Aggregation

```
char aggr[7]="fboaor";
char str1[3], char str2[3];
int i;
for(i=0;i<3;i++){
str1[i]=aggr[i*2];
str2[i]=aggr[i*2+1];
}
```

str1=foo
str2=bar

Ordering

- Mainly reorders the array
- The indices used to access the array can be changed by a function mapping the original position i into its new position

```
if(a[f(i)]>a[f(j)])
   swap(a[f(i)],a[f(j)])
```

Few Points

- A technical way of protecting intellectual property contained within or encapsulated by a software
- When network bandwidth nor latency is an issue then software can be run from a remote server
  - This will prevent to get physical access to the software
- If the end user can be convinced to use tamper resistant hardware, the program can be entirely executed in the hardware using encryption and decryption

Conclusion

- Obfuscation provides certain level of protection
- However, a competent attacker will always be able to reverse engineer a program, given enough time and perseverance
- Obfuscation can make the attack economically inviable
  - As the cost of the attack could outweigh the benefits of a successful attack