

# Obfuscation Techniques

**REVERSE ENGINEERING**

**deti** universidade de aveiro  
departamento de eletrónica,  
telecomunicações e informática

**João Paulo Barraca**

# Obfuscation Techniques

- Aims at hardening the process of reverse engineering
  - Increases level of experience required
  - Increases cost (time, money)
  - Imposes the need for specific tools, techniques and procedures
- Applications (some):
  - **License protected software:** to prevent the generation of arbitrary licenses or subversion of the program code
  - **Proprietary software:** prevent the recovery of a design pattern or algorithm (IP protection)
  - **Malware:** to prevent recovery of the actions, prevent detection, Social Engineer users

# Obfuscation Techniques

## Static vs Dynamic

- Static obfuscation frequently transforms code before execution
  - Maybe before compilation, or during compilation
  - Countering static analysis
  - An obfuscated program is complex to analyze
- Dynamic obfuscation transforms code during execution
  - Countering Dynamic Analysis
  - The obfuscated program may change its behavior, expand or include further code

# Obfuscation Techniques

## Main Categories (Balachandran, TIFS 2013)

- Layout Obfuscation
  - Design Obfuscation
  - Data Obfuscation
  - Control Obfuscation
- 
- Also: Content Type Obfuscation (already addressed)

# Code Obfuscation

## Layout Obfuscation

- Aims at hiding how the source code is structured
  - As source code (or symbols) can present enough information to help reversing a program
- Applied to the source code, and focused on situations where source can be obtained
  - Javascript, HTML, CSS, Java
- Methods:
  - Deleting comments
  - Remove debugging information
  - Renaming classes, methods and variables
  - Removing spaces
  - Stripping a binary

```

define C(c #\
/*size=3173*/#include<stdio.h>
/*crc=b7f9ecff.*/#include<stdlib.h>
/*Mile/Adele_von_Ascham*/#include<time.h>
typedef/**/int(I);I/*:3*/d,i,j,a,b,l,u[16],v
[18],w[36],x,y,z,k;char*P="\n\40()",*,*p,*q,*t[18],m[4];
void/**/O(char*q){for(*q;q++)*q>32?z=111-*q?z=(z+*q)%185,(k?
k--:(y=z%37,(x=z/37%7)?printf(*t,t[x],y?w[y-1]:95):y>14&&y<33?x
=y>15,printf(t[15+x],x?2<<y%16:1,x?(1<<y%16)-1:1):puts(t[y%28]))
,0:z+82:0;}void/**/Q(I(p),I*q){for(x=0;x<p;x++){q[x]=x;}for(--p
=q[x=rand()%~p],q[x]=q[p];}char/**/n[999]=C(Average?!nQVQd%R>Rd%
>1;q[p]=y)y
R% %RNIPRfi#VQ}R;TtuodtsRUD%RUd%RUOSetirwf!RnruterR{RTSniamRtniQ>h.oidts<edulc
ni #V>rebmun<=NIPD-RhtiwRelipmocResaelPRrorre#QNIPRfednfi#V__ELIF__R_
Re nifed#V~-VU0V;}V{R= R][ORrahcRdengisnuRtsnocRcitatsVesle#Vfidne#V53556
. .1RfoRegnarRehtRniRre getniRnaRsiR]NIP[R erehwQQc.tuptuoR>Rtxt.tupniR
< R]NIP[R:egasuV_Redulcn i#VfednfiVfednuVenife dVfedfiVQc%Rs%#V);I/**/main(
I( f),char**e){if(f){for(i= time(NULL),p=n,q= n+998,x=18;x;p++){*p>32&&!(
*--q=*p>80&&*p<87?P[*p- 81]:* p)?t [( -- x)]=q+1;q;}if(f-2||d=atoi
(e[1]))<1||65536<d){;0(" \"); goto 0;}srand(i);Q(16,u);i=0;Q(
36,w);for(;i<36; i++){w[i] +=w [i]<26 ? 97:39; }O(C(ouoo9oBotoo%#
ox^#oy_#ozoou#o{ a#o|b#o}c# o~d#oo-e #oo. f#oo/g#oo0h#oo1i#oo
2j#oo3k#oo4l#o p));for(j =8;EOF -(i= getchar());l+=1){a=1+
rand()%16;for(b =0;b<a||i- main (0,e);b++)x=d^d/4^d/8^d/
32,d= (d/ 2|x<<15)&65535; b|= !l<<17;Q(18,v);for(a=0;a<18;
a++ ){if( (b&(1<<(i=v[a] )))*) m=75+i,0(m),j=i<17&&j<i?i:j;}O(C(
!) ); }O(C(oqovoo97o /n!));i= 0;for(;i<8;O(m))m[2]=35,*m=56+u[i],m[1
]= 75 +i++;O(C(oA!oro oqoo9) );k=112-j*7;O(C(6o. !Z!Z#5o- !Y!Y#4~!X!X#3}
!W !W #2 |!V!V#1{!U!U#0z! T!T#/y!S!S#.x!R!R#-w!Q!Q#ooAv!P!P#+o#!O!O#*t!N!
N# oo >s!M!M#oo=r!L!L#oo<q!K!K# &pIo@:;= oUm#oo98m##oo9=8m#oo9oUm###oo9;=8m#o
o9 oUm##oo9=oUm#oo98m#### o09] #o1:^#o2;_#o3<o ou#o4=a#o5>b#o6?c#o
7@d#o8A e#o 9B f#o:Cg#o; D h#o<Ei #o=Fj#o> Gk#o?Hl#oo9os#####
));d=0 ;} 0: for(x=y=0;x<8;++
x)y|= d&(1<<u[x])?
1<< x:0;return
/* :9 */
y ; }

```

# Code Obfuscation

## Design Obfuscation

- Aims at making the design nonobvious, more difficult to recover
  - Usually done by a tool before compilation or during compilation
  - GCC can do this automatically by inlining functions (`-O3 -finline -funroll-loops`)
- Methods:
  - Merging and splitting methods
  - Merging and splitting classes
  - Splitting binary code, while inserting dummy instructions
  - Splitting loops and conditions, maybe interleaved with dummy code
  - Inlining functions
  - Dead Code

# Code Obfuscation

## Design Obfuscation – Breaking Code

Code inserted, but never executed.  
JMP before dummy code effectively only splits code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  unsigned long long factorial(unsigned long long a) {
5
6      unsigned long long r = 1;
7
8      while(a > 0){
9          unsigned long long v = r * a;
10         if(v < r){
11             printf("ERROR: Overflow\n");
12             exit(-1);
13         }
14         r = v;
15         a = a - 1;
16     }
17     return r;
18 }
19
20 int main(int argc, char** argv) {
21     unsigned long long v = 0;
22     if(argc != 2) {
23         printf("Need a positive integer argument\n");
24         return -1;
25     }
26     v = atol(argv[1]);
27
28     if(v <= 0){
29         printf("Need a positive integer argument\n");
30         return -1;
31     }
32
33     printf("Result: %llu\n", factorial(v));
34
35     return 0;
36 }
```

```
21  int main(int argc, char** argv) {
22     unsigned long long v = 0;
23     if(argc != 2) {
24         printf("Need a positive integer argument\n");
25         return -1;
26     }
27     asm("jmp label");
28     factorial(factorial(argc));
29     asm("label:");
30     v = atol(argv[1]);
31
32     if(v <= 0){
33         printf("Need a positive integer argument\n");
34         return -1;
35     }
36
37     asm("jmp label_b");
38     factorial(factorial(v * factorial(-v)));
39     asm("label_b:");
40
41     printf("Result: %llu\n", factorial(v));
42
43     return 0;
44 }
45 }
```

# Code Obfuscation

## Design Obfuscation – Breaking Code

Code inserted, but never executed.  
**JMP** before dummy code effectively only splits code

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 unsigned long long factorial(unsigned long long a) {
5
6     unsigned long long r = 1;
7
8     while(a > 0){
9         unsigned long long v = r * a;
10        if(v < r){
11            printf("ERROR: Overflow\n");
12            exit(-1);
13        }
14        r = v;
15        a = a - 1;
16    }
17    return r;
18 }
19
20 int main(int argc, char** argv) {
21     unsigned long long v = 0;
22     if(argc != 2) {
23         printf("Need a positive integer argument\n");
24         return -1;
25     }
26     v = atol(argv[1]);
27
28     if(v <= 0){
29         printf("Need a positive integer argument\n");
30         return -1;
31     }
32
33     printf("Result: %llu\n", factorial(v));
34
35     return 0;
36 }
```

```
21 int main(int argc, char** argv) {
22     unsigned long long v = 0;
23     if(argc != 2) {
24         printf("Need a positive integer argument\n");
25         return -1;
26     }
27 }
```

What about the output binary?

Compile with `gcc -O0 -o factorial-split factorial-split.c`

Does it effect static or dynamic analysis?

Check with `objdump -d` and `ghidra`

What about if instead of `jmp` you use `jz` or `jnz`?

`gcc` may also inline functions (the opposite) when using `-O3` or `-finline-functions`

# Code Obfuscation

## Design Obfuscation – Dead Code

- Aims at inserting dummy code to confuse the analysis
  - Code may follow some pattern (previous example), or be random
  - Code may lock the analysis tool if recursive disassembly is used
  - Decompilation to Pseudo C will surely be affected
- Dead code can be added after compilation
  - May contain fingerprinting information by making binaries unique

# Code Obfuscation

## Design Obfuscation – Dead Code

```
21 unsigned long long factorial(unsigned long long a) {
22
23     unsigned long long r = 1;
24
25     while(a > 0){
26         unsigned long long v = r * a;
27         if(v < r){
28             printf("ERROR: Overflow\n");
29             exit(-1);
30         }
31         r = v;
32         a = a - 1;
33
34         if(v != r) {
35             __asm__ (REP(3,3,3,"nop;"));
36         }
37     }
38     return r;
39 }
```

$r=v$ , therefore, `if(v!=r)` will be always false. Compiler will not easily discard this code.

`__asm__ . . . .` Instruction will insert 333 NOPs (which will not be executed)

This is a placeholder that can be used later for post processing by editing the binary directly

# Exercise

## Design Obfuscation – Dead Code

- Consider the example presented in factorial-dead.c
- Create a script that replaces a sequence of NOP (0x90) with another sequence of instructions
- Take care that in X86, instructions are of variable length. The last 15 instructions added to a placeholder should be handled with care, as they can “mask” real instructions after the placeholder
  - For this exercise, leave them with 0x90
- Compile the code and then use **objdump** and **ghidra** to reverse the binary. Is this useful? Can it be detected?

# Code Obfuscation

## Design Obfuscation – Dead Code

```
2 undefined8 main(int param_1, long param_2)
3
4 {
5     undefined8 uVar1;
6     long lVar2;
7
8     if (param_1 == 0x2) {
9         lVar2 = atol(*(char **)(param_2 + 0x8));
10        if (lVar2 == 0x0) {
11            puts("Need a positive integer argument");
12            uVar1 = 0xffffffff;
13        }
14        else {
15            uVar1 = factorial(lVar2);
16            printf("Result: %llu\n", uVar1);
17            uVar1 = 0x0;
18        }
19    }
20    else {
21        puts("Need a positive integer argument");
22        uVar1 = 0xffffffff;
23    }
24    return uVar1;
25 }
26
```

C: Decompile: main - (factorial-dead-obf)

```
13 undefined4 *local_28;
14 int local_1c;
15 long local_10;
16
17 local_10 = 0x0;
18 local_28 = param_2;
19 local_1c = param_1;
20 if (param_1 == 0x2) {
21     puVar4 = *(undefined4 **) (param_2 + 0x2);
22     uStack48 = 0x10136a;
23     local_10 = atol((char *)puVar4);
24     if (local_10 == 0x0) {
25         uStack48 = 0x101381;
26         puts("Need a positive integer argument");
27         pcVar2 = (char *)0xffffffff;
28     }
29     else {
30         if (local_1c * local_10 == 0x0) {
31             *puVar4 = *param_2;
32             if ((POPCOUNT(local_1c * local_10 & 0xff) & 0x1U) != 0x0) {
33                 /* WARNING: Bad instruction - Truncating control flow here */
34                 halt_baddata();
35             }
36             puVar4 = (undefined4 *) (ulong) ((int)param_4 - 0x44);
37             puVar3 = &uStack48;
38             cVar1 = '\x12';
39             do {
40                 puVar4 = puVar4 + -0x1;
41                 puVar3 = (undefined8 *) ((long)puVar3 + -0x4);
42                 *(undefined4 *)puVar3 = *puVar4;
43                 cVar1 = cVar1 + -0x1;
44             } while ('\0' < cVar1);
45             /* WARNING: Bad instruction - Truncating control flow here */
46             halt_baddata();
47         }
48         uStack48 = 0x10172f;
49         factorial(local_10);
50         uStack48 = 0x101743;
51         printf("Result: %llu\n");
52         pcVar2 = (char *) (local_1c * local_10);
53         if (pcVar2 + -(local_10 + -0x3) != NULL) {
54             pcVar2 = NULL;
55         }
56     }
57 }
```

# Code Obfuscation

## Data Obfuscation

- Encrypts, or otherwise encodes data contents
  - Contents are decrypted in real time, as the program is executed
  - Static analysis, or fingerprint matching may fail to correctly recover useful information
  - Frequent tactic to evade filters
- Why?
  - Strings frequently carry semantic information, that may help analysis
  - E.g. Str="Please input your AES key": we will know that this a key, and know the algorithm

# Code Obfuscation

## Data Obfuscation - how

- Split the string in parts
  - May be combined with two conditions or loops to validate both parts individually
- Erase strings right after use
- Common XOR is frequently found as it requires no dependencies and is fast
  - More recent malware will use RC4 or even AES for this purpose
  - Decryption key can also be encrypted, and some key may be obtained dynamically
    - E.g. from a hardware token as a form of licensing enforcement
- Create a custom encoding based on a complex state machine
  - May use flow information, voiding the decoding of strings if the execution order it changed

# Code Obfuscation

## Control Obfuscation

- Introduces dummy control structures, with little impact to execution
  - Impact is only from a performance point of view
  - However, analysis tools will interpret the control structures and create complex CFGs
- Makes use of Opaque Predicates: predicates for which the programmer already knows the result.
  - E.g. `if ( 1 > 0 )`

# Code Obfuscation

## Control Obfuscation – Opaque Predicates

- Introduces dummy control structures, with little impact to execution
  - Impact is only from a performance point of view (additional branch)
  - However, analysis tools will interpret the control structures and create complex CFGs
  
- Makes use of Opaque Predicates: predicates for which the programmer already knows the result.
  - E.g. `if ( 1 > 0 )` or `v=r; if(v==r)`

# Code Obfuscation

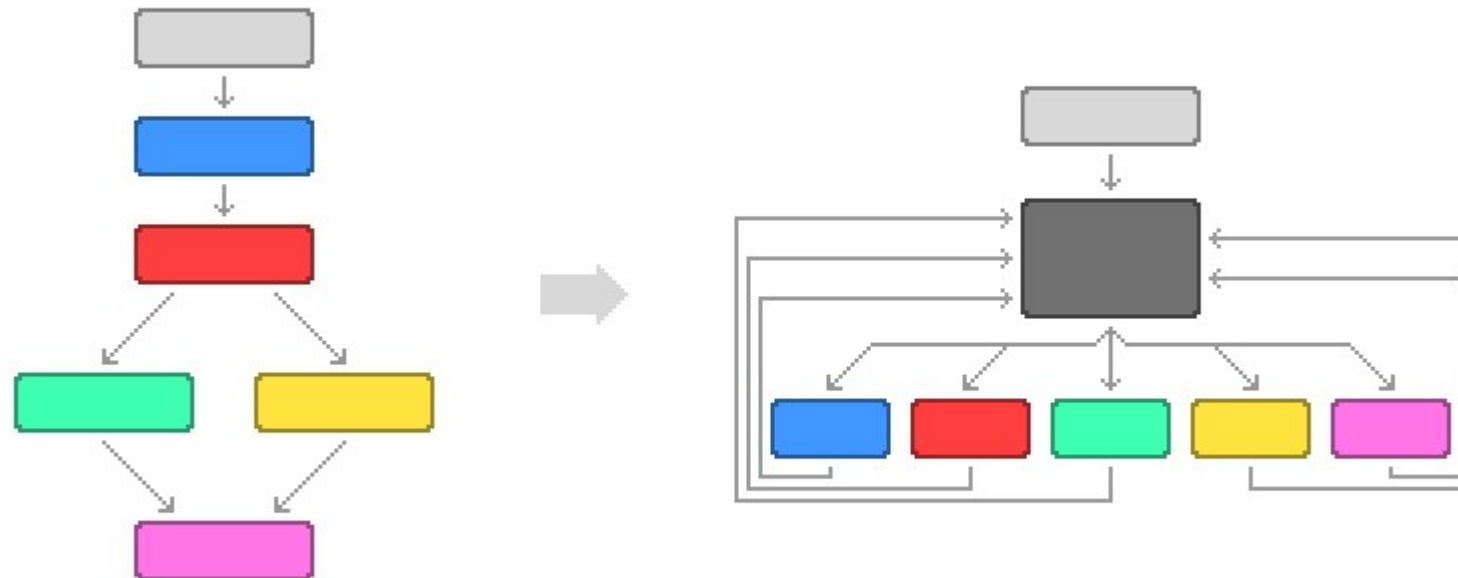
## Control Obfuscation – Opaque Predicates

- Opaque predicates can be more complex
- Manipulate pointers, linked lists, use computation processes
- Result of a predicate can be dynamic, and related to execution state
  - Dynamic analysis may change execution sequence, therefore the predicate result and invalidate the execution
  - Similar to TPMs, where keys are provided at a valid situation
  - Predicate can use dynamic data, received from external services
- Concurrency can be used to create predicates
  - If two threads are executing with some relation, one can update data, that the other uses to construct a predicate
  - Timing information can also be used, to further increase the complexity (information not available statically)

# Code Obfuscation

## Control Obfuscation – Control Flow Flattening

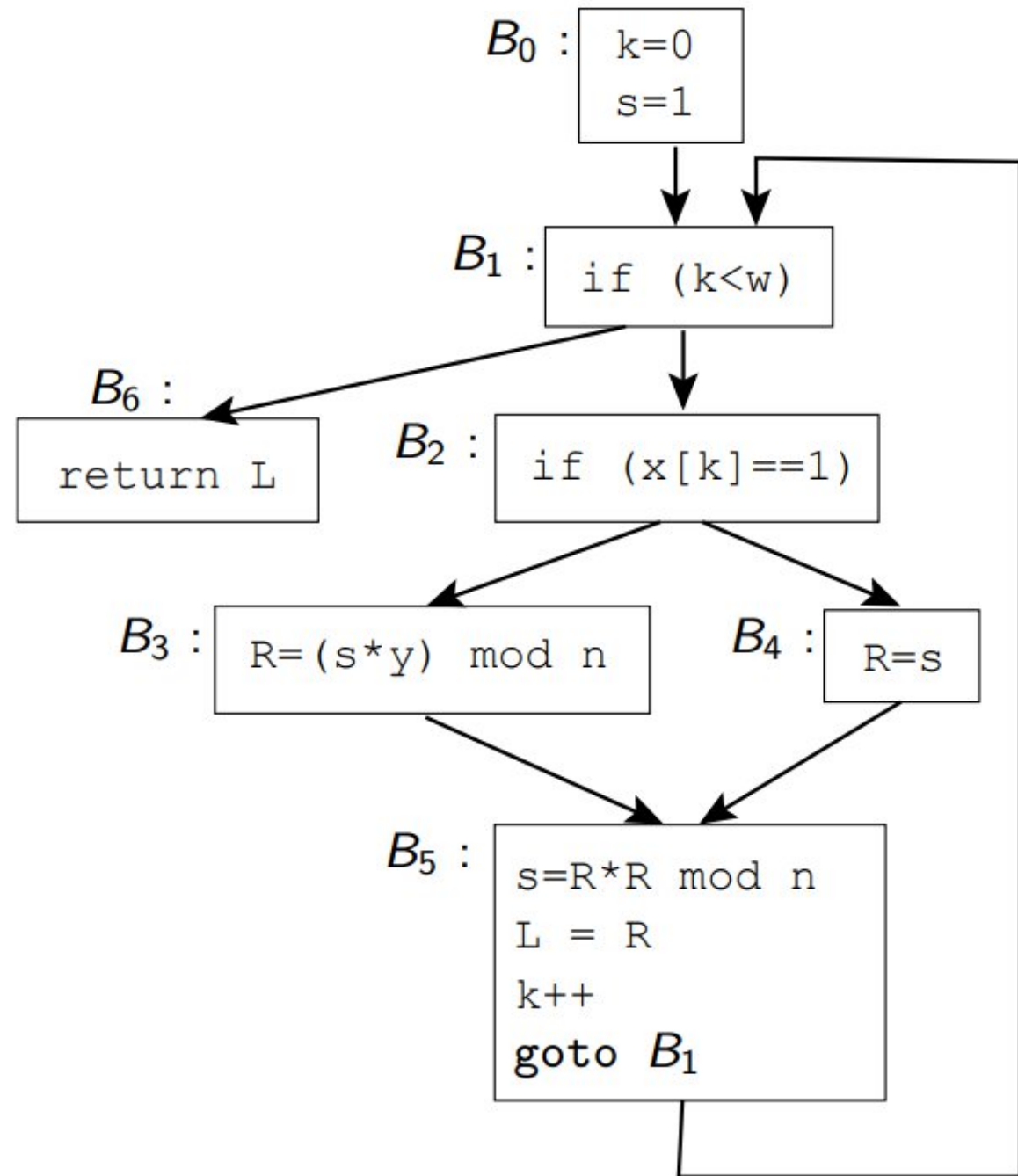
- Removes control flow structures from program
  - Converts the program to a gigantic Switch, where each condition is a case
  - Program runs on an infinite loop around the switch
- Program becomes ~4 times slower, and 2 times larger



```

int modexp(int y,int x[],
           int w,int n) {
    int R, L;
    int k = 0;
    int s = 1;
    while (k < w) {
        if (x[k] == 1)
            R = (s*y) % n;
        else
            R = s;
        s = R*R % n;
        L = R;
        k++;
    }
    return L;
}

```



```
int modexp(int y, int x[], int w, int n) {
    int R, L, k, s;
    int next=0;
    for(;;)
        switch(next) {
            case 0 : k=0; s=1; next=1; break;
            case 1 : if (k<w) next=2; else next=6; break;
            case 2 : if (x[k]==1) next=3; else next=4; break;
            case 3 : R=(s*y)%n; next=5; break;
            case 4 : R=s; next=5; break;
            case 5 : s=R*R%n; L=R; k++; next=1; break;
            case 6 : return L;
        }
}
```

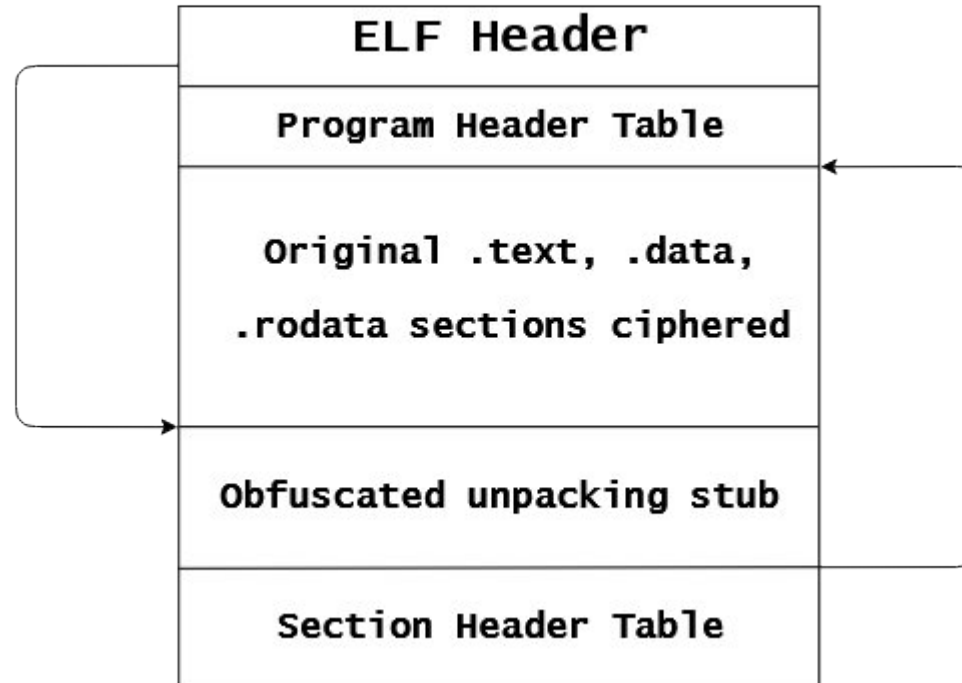
# Code Obfuscation

## Self Decompressing Binaries

- What is a Self-Decompressing Binary (Packer)?
  - An executable that contains a compressed or encrypted version of the original program.
  - It includes a small piece of code called a **Stub** responsible for extracting the payload into memory at runtime.
- Primary Use Cases:
  - **Legitimate**: Reducing file size for distribution, protecting intellectual property (DRM).
  - **Malicious/Obfuscation**: Hiding malicious payloads from Antivirus (AV) scanners, defeating static analysis, and slowing down reverse engineers.
- Packer is the generic term used with these

# Code Obfuscation

## Self Decompressing Binaries



# The Architecture of a Packed Binary

## How a Packer Alters a File

- **Original Executable:** Contains standard **.text** (code), **.data** (variables), and an Original Entry Point (OEP).
- **Packed Executable:** The original sections are compressed and or encrypted into a generic data section.
- **The Stub:** A new executable section is added.
  - The OS Entry Point is modified to point to this Stub.

# Windows Implementation: The PE Format

## Self Decompressing Binaries: Linux

- **Section Manipulation:** Packers like UPX replace standard PE sections (like `.text`, `.rdata`) with custom ones (e.g., `UPX0`, `UPX1`).
- **The IAT Problem:** Lloader uses the Import Address Table (IAT) to link DLLs.
  - A packed binary hides its true imports.
- **Stub Responsibilities:** The stub must manually act as the Windows Loader.
  - It uses `LoadLibrary` and `GetProcAddress` to reconstruct the true IAT in memory before jumping to the OEP.
- **Common Windows Packers:** UPX, VMProtect, Themida, Armadillo.

# Linux Implementation: The ELF Format

- **Memory Management:** The ELF loader maps segments into memory.
  - The unpacking stub takes control via the modified ELF entry point.
- **Syscalls for Execution:** The stub typically uses `mmap`
  - Allocate memory with `PROT_WRITE` to unpack the data
  - Uses `mprotect` to change the permissions to `PROT_EXEC` (execution) and `PROT_READ`.
- **Advanced Techniques:** Modern Linux packers are fileless
  - malware might use `memfd_create` (creating an anonymous file in RAM) or execute directly via `/proc/self/mem` to avoid touching the disk entirely.

## How To Defeat Packers

- **Static Unpacking:** Reverse engineering the stub's decryption algorithm, write a custom script to unpack the binary.
  - Safe but highly difficult.
- **Dynamic Unpacking:**
  - Run the malware in a controlled debugger (x64dbg / GDB).
  - Find the Tail Jump (the jump from the stub to the Original EP).
  - Set a hardware breakpoint on memory execution (HW BPX).
  - Once the breakpoint hits, the program is unpacked in RAM.
  - Dump the memory to disk and reconstruct the IAT/Imports using tools like Scylla.