Systems Security
Ruhr-University Bochum

Program Analysis

Lecture 06: *Reconstructing Information II*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Next exercise will be published next week
- Feedback on second exercise?
- Next week we have a small change
  - Exercise from 9:00 to 10:30 (Ralf Hund)
  - Lecture from 10:30 to 12:00 (Carsten Willems)

Announcements

RUHR-UNIVERSITÄT BOCHUM

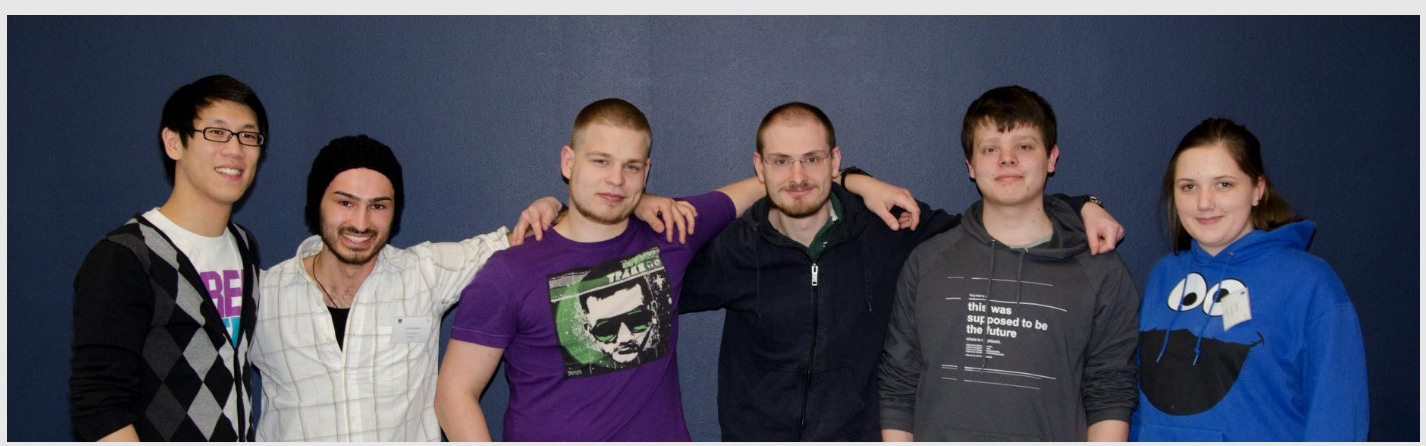
hgi
Horst Görtz Institut
für IT-Sicherheit

RUB

ITS.Botschafter
Infotreffen

Dienstag, der 22. November 2011
15 Uhr
Raum ID 2 / 404

Anmeldung: www.hgi.rub.de



ITS.Botschafter HORST GÖRTZ INSTITUT FÜR IT-SICHERHEIT | Bochum | 17.11.2011 1

Last Week

- Recognizing loops
- Finding data structures
- Optimization
  - Constant propagation
  - Dead code elimination
  - Inlining

Loop Optimization

- Programs spend a lot of time in loops, thus it makes sense to optimize them as good as possible
- Leads to some common constructs by compilers that we need to understand when analyzing code:
  - Unswitching
  - Loop unrolling
  - Loop inversion
  - ...

Unswitching

Original

```
for (i=0; i<1000; ++i) {
  if (!a)
  { /*.1.*/ }
  else
  { /*.2.*/ }
}
```

Unswitching

Original	Unswitched

<pre>for (i=0; i<1000; ++i) {
  if (!a)
  { /*.1.*/ }
  else
  { /*.2.*/ }
}</pre>
	<pre>if (!a) {
	  for (i=0; i<1000; ++i)
	    { /*.1.*/ }
	}
	else {
	  for (i=0; i<1000; ++i)
	    { /*.2.*/ }
	}</pre>

If the variable a is not changed within the loop, then the conditional will *always* evaluate the same way for each loop iteration

⇒ By unswitching, 999 comparisons are saved

Loop Unrolling

Without Unrolling	With Unrolling

`for(int i = 0; i < 100; i++) {`
` function(i);`
`}`
	`for(int i = 0; i < 100; i+=4)`
	`{`
	` function(i);`
	` function(i+1);`
	` function(i+2);`
	` function(i+3);`
	`}`

- Unrolled version is faster since there are fewer conditional jumps
- Compiler tries to avoid conditional jumps since this leads to faster code in general

Loop Inversion

Which loop is faster?

Loop A	Loop B
`while (x < y) { ... }`	`if (x < y) { do { ... } while (x < y); }`

Loop A

```
while (x < y) {
  ...
}
```

Loop B

```
if (x < y) {
  do {
    ...
  } while (x < y);
}
```

Loop Inversion

Which loop is faster?

Loop A

```
while (x < y) {
  ...
}
```

Loop B

```
if (x < y) {
  do {
    ...
  } while (x < y);
}
```

```
    cmp [edi+60h], ebx
    jz  short return

; loop body omitted

do_while_check:

    inc ebx
    cmp ebx, [edi+60h]
    jnz short loop_body

return:
```

- Loop B is a bit faster, less jumps (last iteration)
- Structure of loops can be changed by compiler

Control Flow Optimizations

- These optimizations improve the control-flow structure for a function in various ways
- Several examples
  - Branch-to-branch elimination
  - Sub-expression elimination
  - Branchless code

Branch-To-Branch Elimination


```
while(cond1)
{
   if(cond2)
   {
      if(cond3)
      {
         /* #1 */
      }
   }
   else
   {
      /* #2 */
   }
}
```

```
while_header:
if(!cond1)
  goto follow_while;

if(!cond2)
  goto do_else;

if(!cond3)
  goto follow_else;

/* #1 */
goto follow_else;

do_else:
/* #2 */
follow_else:
goto while_header;

follow_while:
```

Branch-To-Branch Elimination


```
while(cond1)
{
   if(cond2)
   {
      if(cond3)
      {
         /* #1 */
      }
```

The red arrows can instead point at the black target, instead of a branch that takes them there.

```
while_header:
if(!cond1)
  goto follow_while;

if(!cond2)
  goto do_else;

if(!cond3)
  goto follow_else;

/* #1 */
goto follow_else;

do_else:
/* #2 */
follow_else:
goto while_header;

follow_while:
```

Sub-Expression Elimination

The values need only be computed once:

```
int e = b + c + d ;
int f = a + c + d ;

     Struct2->Struct1->Member1;
     Struct2->Struct1->Member2;

int a = x * y + 4 ;
int b = x * y + 7 ;
```

sbb Instruction

- The `sbb` instruction subtracts the two operands, like `sub` would, and then subtracts the carry flag (0 or 1)
- Can be used for evaluating conditionals without branches

Operation

DEST ← (DEST - (SRC + CF));

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

sbb Instruction

- The `sbb` instruction subtracts the two operands, like `sub` would, and then subtracts the carry flag (0 or 1)
- Can be used for evaluating conditionals without branches

```
cmp     dword ptr [ecx+784h], 2
sbb     eax, eax
inc     eax
```

```
eax = (ecx.f784 >= 2);
```

Branchless Code

Substitute conditional jumps with semantic equivalent code that does not need jumps

C Code

```
if(x != 0)
  x = 14;
else
  x = 71;
```

Branchless Code

Substitute conditional jumps with semantic equivalent code that does not need jumps

C Code

`if(x != 0)`
`  x = 14;`
`else`
`  x = 71;`

Assembler	x = 0	x != 0
`neg edi`	0	...
`sbb edi, edi`	0	-1
`and edi, 0FFFFFFC7h`	0	0FFFFFFC7h
`add edi, 47h`	47h(71)	0Eh(14)

Branchless Code

Substitute conditional jumps with semantic equivalent code that does not need jumps

Operation

```
IF DEST = 0
    THEN CF ← 0;
    ELSE CF ← 1;
FI;
DEST ← [- (DEST)]
```

C Code

```
if(x != 0)
  x = 14;
else
  x = 71;
```

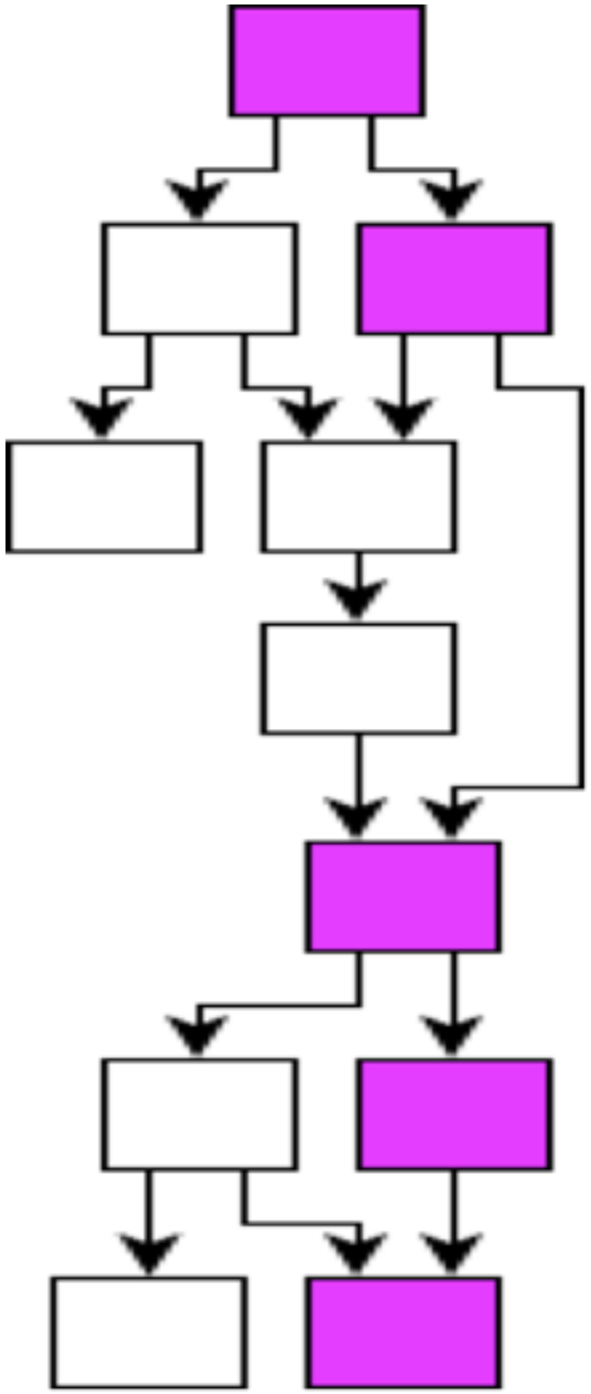
Assembler	x = 0	x != 0
`neg edi`	0	…
`sbb edi, edi`	0	-1
`and edi, 0FFFFFFC7h`	0	0FFFFFFC7h
`add edi, 47h`	47h(71)	0Eh(14)

Frame Pointer Omission

- Instead of using `ebp` as a frame pointer, the compiler may simply use displacements off of `esp` to access local variables and arguments
- `ebp` can then be used as a general purpose register

Hot and Cold Parts

- Suppose magenta paths are most commonly taken (“hot part”)
- We want to keep this code “hot”
- Take advantage of caching and similar effects
- White blocks (“cold parts”) can be placed elsewhere



Hot and Cold Parts

- Split functions into sets of "hot" and "cold" *basic blocks*, causes *function chunking*
- Sort by frequency of execution.
  - Memory pages consist of portions of code with roughly the same likelihood of being executed
- If the OS needs to trim the process' memory, the least-likely-to-execute code will be paged out first
- Reduces "page thrashing", enhances performance

Hot Part

```
cmp     _LdrpShutdownInProgress, 0
mov     [ebp+var_4], edi
jnz     loc_7C95B390
cmp     [ebp+arg_0], edi
jz      loc_7C95B397
push    esi
mov     esi, [ebp+arg_4]
cmp     esi, edi
jz      loc_7C95B3A1
lea     eax, [ebp+arg_0]
push    eax
push    edi
call    _RtlpCaptureImpersonation@8
cmp     eax, edi
jl      ret_pop_esi
push    ebx
lea     ebx, [esi+10h]
mov     eax, 0B000h
lock or [ebx], eax
cmp     [ebp+arg_8], 0FFFFFFFFh
jz      loc_7C93E770

loc_7C93D75D:
```

Cold Part, on different memory pages

```
loc_7C95B390:
                xor     eax, eax
                jmp     ret_pop_edi
; ---------------------------------------

loc_7C95B397:
                mov     eax, 0C00000EFh
                jmp     ret_pop_edi
; ---------------------------------------

loc_7C95B3A1:
                mov     eax, 0C00000F0h
                jmp     ret_pop_esi
```

```
loc_7C93E770:
                call    _RtlpGetWaitEvent@0
                cmp     eax, edi
                mov     [ebp+var_4], eax
                jz      loc_7C95B3AB
                jmp     loc_7C93D75D
```

Obfuscation

Motivation

- Last chapter focussed on *optimization*: how does a compiler transform a piece of code?
  - Code optimization
  - Loop optimization
  - Control flow optimization
- All of these techniques make analysis harder
- What happens if an attacker adds obfuscation *on purpose* to make our life harder?

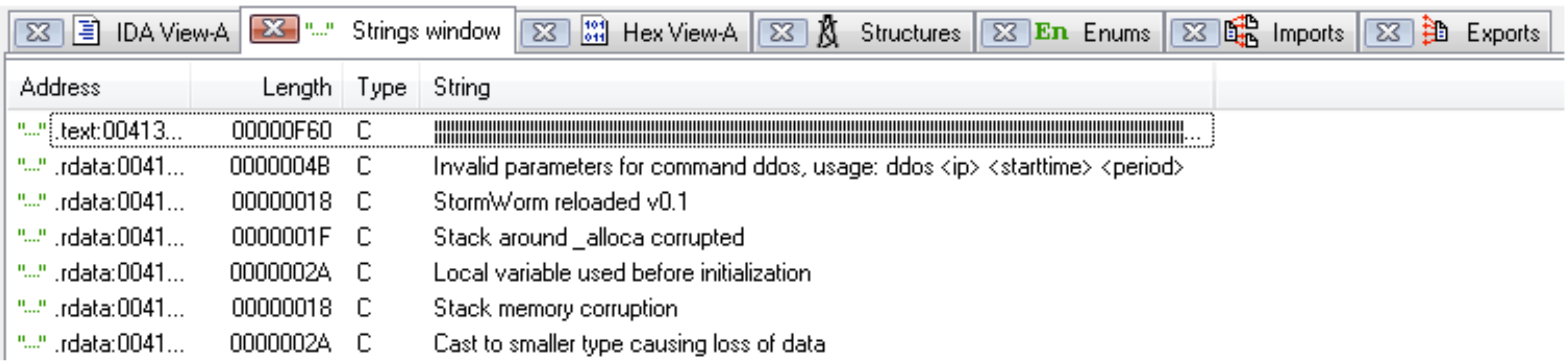
Motivation

Source: http://www.econsteve.com/?p=156

- Many methods to obfuscate code, for example
  - (Weak) Encryption
  - Complex transformations
  - Hide relevant information (*needle in a haystack*)
  - Add bogus code
  - ...

String Obfuscation

- Strings embedded in binary often help to understand what a given piece of code does

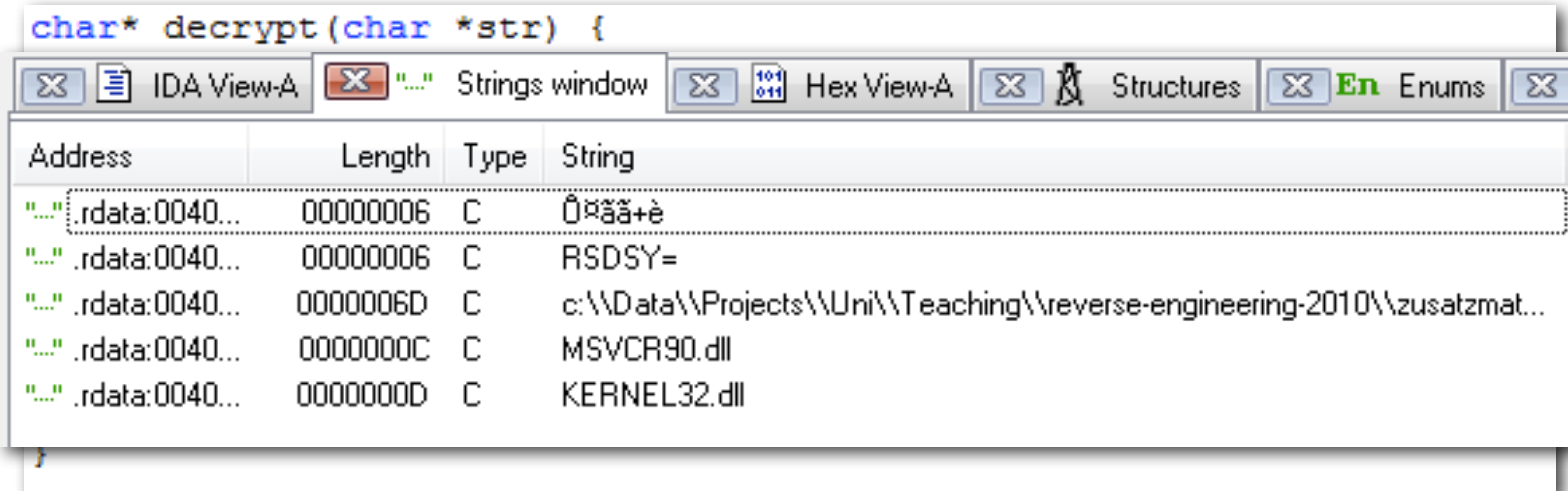


- An attacker can hide/obfuscate these strings
- Decoding during runtime (often simple XOR)

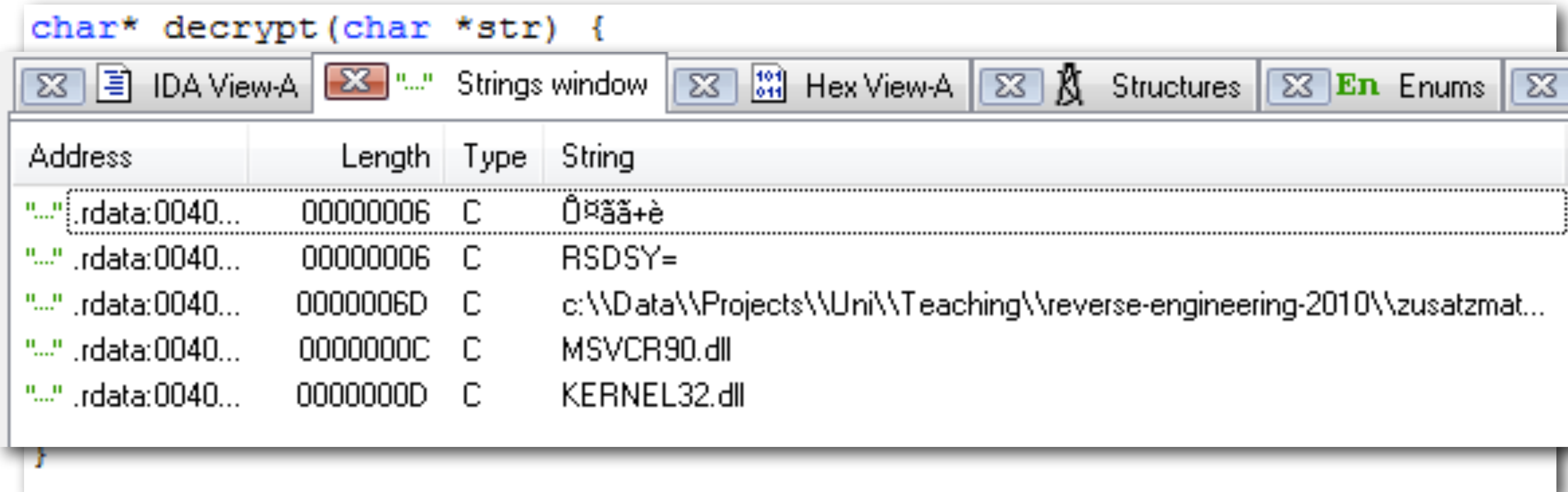
Example

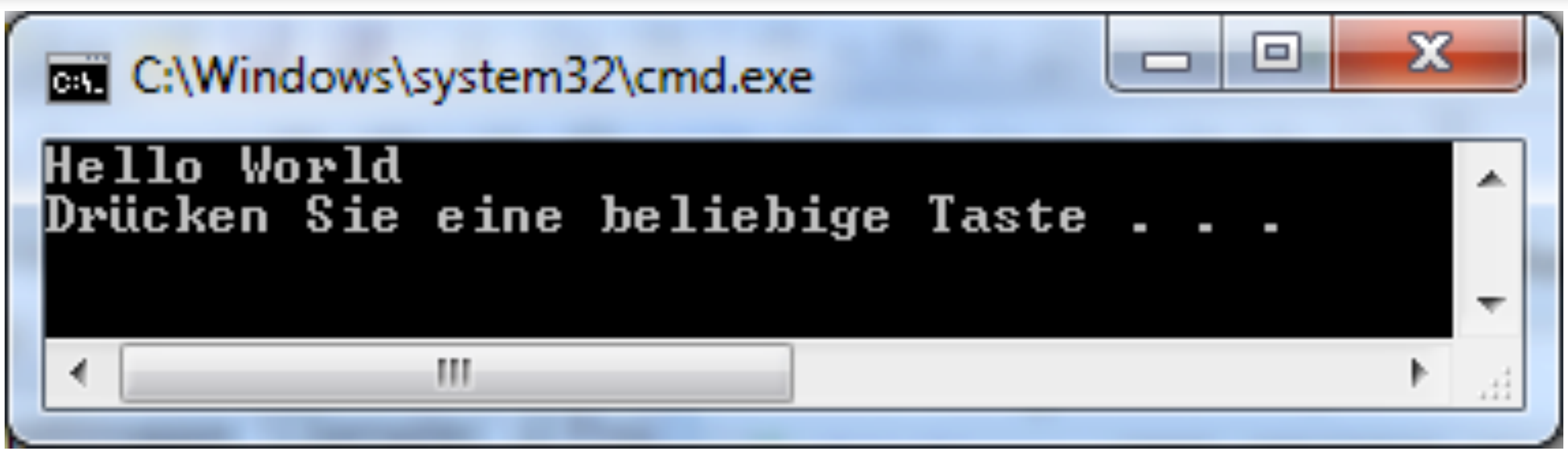

```c
char* decrypt(char *str) {
    for(unsigned int i = 0; i < strlen(str); i++)
        str[i] ^= 0xAA;
    return str;
}

int main(int argc, char *argv[]) {
    char str[] = "\xE2\xCF\xC6\xC6\xC5\x8A\xFD\xC5\xD8\xC6\xCE\xA0";
    printf(decrypt(str));
}
```

Example


char* decrypt(char *str) {
IDA View-A
"..." Strings window
Hex View-A
Structures
En Enums
Address
Length
Type
String
"..." .rdata:0040... 00000006 C Ô¤ãã+è
"..." .rdata:0040... 00000006 C RSDSY=
"..." .rdata:0040... 0000006D C c:\\Data\\Projects\\Uni\\Teaching\\reverse-engineering-2010\\zusatzmat...
"..." .rdata:0040... 0000000C C MSVCR90.dll
"..." .rdata:0040... 0000000D C KERNEL32.dll
}

Example


char* decrypt(char *str) {
IDA View-A
Strings window
Hex View-A
Structures
En Enums
Address
Length
Type
String
.rdata:0040... 00000006 C Ô¤ãã+è
.rdata:0040... 00000006 C RSDSY=
.rdata:0040... 0000006D C c:\\Data\\Projects\\Uni\\Teaching\\reverse-engineering-2010\\zusatzmat...
.rdata:0040... 0000000C C MSVCR90.dll
.rdata:0040... 0000000D C KERNEL32.dll


C:\Windows\system32\cmd.exe
Hello World
Drücken Sie eine beliebige Taste . . .

Junk Code

- An attacker can add useless instructions which do not change the semantics of a program
    - Complementary to *dead code elimination*
    - “Semantical NOP”
- Makes it harder to analyze and understand code
- Cumbersome analysis of each instruction
    - Is the result of an operation used after all?
    - There are automated solutions to deal with this

Junk Code

Example I

```
add eax, 1
```

Example II

```
add eax, 1
```

Example III

```
add eax, 1
```

Junk Code I

```
add eax, 3      ; note that this might
sub eax, 3      ; cause problems
add eax, 1
sub eax, 0
```

Junk Code

Example I

```
add eax, 1
```

Junk Code I

```
add eax, 3      ; note that this might
sub eax, 3      ; cause problems
add eax, 1
sub eax, 0
```

Example II

```
add eax, 1
```

Junk Code II

```
push esi
add eax, 1
pop esi
```

Example III

```
add eax, 1
```

Junk Code

Example I

```
add eax, 1
```

Junk Code I

```
add eax, 3      ; note that this might
sub eax, 3      ; cause problems
add eax, 1
sub eax, 0
```

Example II

```
add eax, 1
```

Junk Code II

```
push esi
add eax, 1
pop esi
```

Example III

```
add eax, 1
```

Junk Code III

```
mov esi, [ebp+8]    ; no access to esi
inc ebx             ; afterwards
add eax, 1
dec ebx
```

Code Permutation

- "There's more than one way to do it"
- A programmer can express a specific statement in many different ways, for example
  - `mov eax, 0`
  - `xor eax, eax`
  - `sub eax, eax`
- An attacker can use this to obfuscate her code

Code Permutation

<u>Original:</u>

```
mov [X], 42
```

Code Permutation

Original:

```
mov [X], 42
```

Permutation 1:

```
push 42
pop [X]
```

Permutation 2:

```
mov eax, X
mov [eax], 42
```

Code Permutation

Original:

```
mov [X], 42
```

Permutation 1:

```
push 42
pop [X]
```

Permutation 2:

```
mov eax, X
mov [eax], 42
```

Permutation 3:

```
mov edi, X
mov eax, 42
stosd
```

Permutation 4:

```
push X
pop edi
push 42
pop eax
stosd
```

Code Permutation

STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Description:

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation).

Code Permutation

Original:

```
mov [X], 42
```

Permutation 1:

```
push 42
pop [X]
```

Permutation 2:

```
mov eax, X
mov [eax], 42
```

Permutation 3:

```
mov edi, X
mov eax, 42
stosd
```

Permutation 4:

```
push X
pop edi
push 42
pop eax
stosd
```

Other Techniques

Code reordering: Instructions that are indepent from each other are reordered to obfuscate flow

Example	Reordered Code

`mov eax, [x]`
`mov ebx, [y]`
`sub eax, ebx`
`mov [x], eax`
	`mov ebx, [y]`
	`mov eax, [x]`
	`sub eax, ebx`
	`mov [x], eax`

Replace code with semantically equivalent code

Example	Equivalent Code

`mov eax, [x]`
`mov ebx, [y]`
`sub eax, ebx`
`mov [x], eax`
	`xor eax, eax`
	`add eax, [x]`
	`sub eax, [y]`
	`mov [x], eax`

Morphing Code

Polymorphism

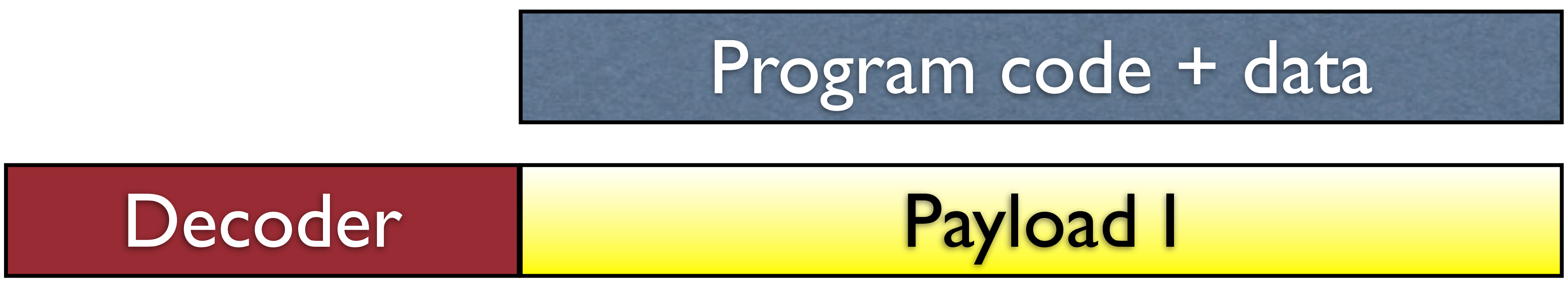
- A piece of code is *polymorph* if it is able to changes its own structure
  - Typically found in malware or heavily obfuscated code
- Each copy of the binary has a different byte pattern or structure
  - Makes analysis harder
  - Makes detection harder, especially for antivirus companies (no static signatures)

Polymorphism

- Typically a small stub is constant for each iteration of polymorphic code
- This is typically a *decoder/decrypter* that is able to change the structure of the code
  - For example, change instructions without changing the semantic meaning
  - Or change the key used to decrypt the code
- If even the stub is changing all the time we call a piece of code *metamorph*

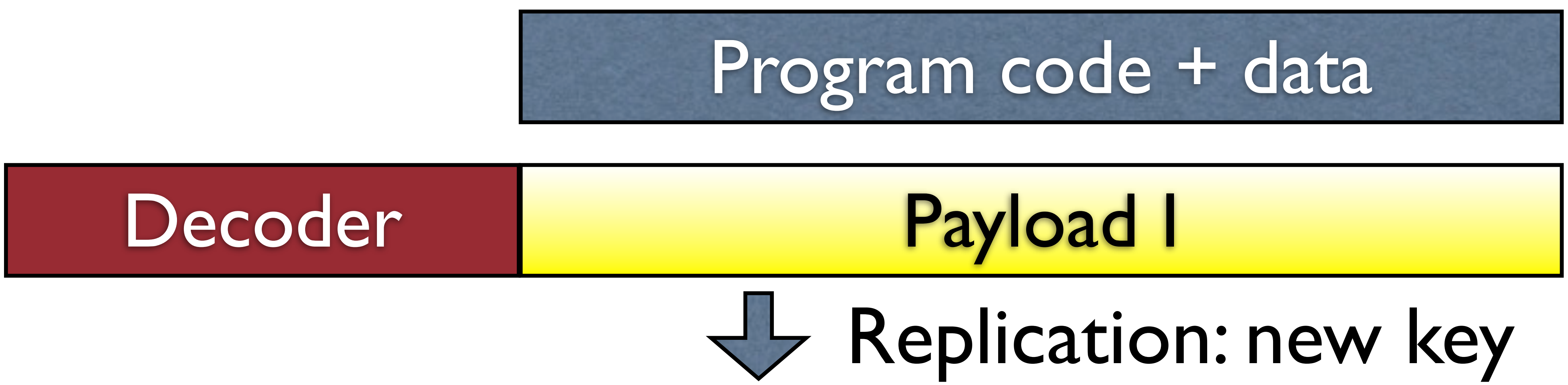
Polymorphism

- Program code and data are encoded
- Decoder prepares the program and then executes it
- Key is changed in each iteration



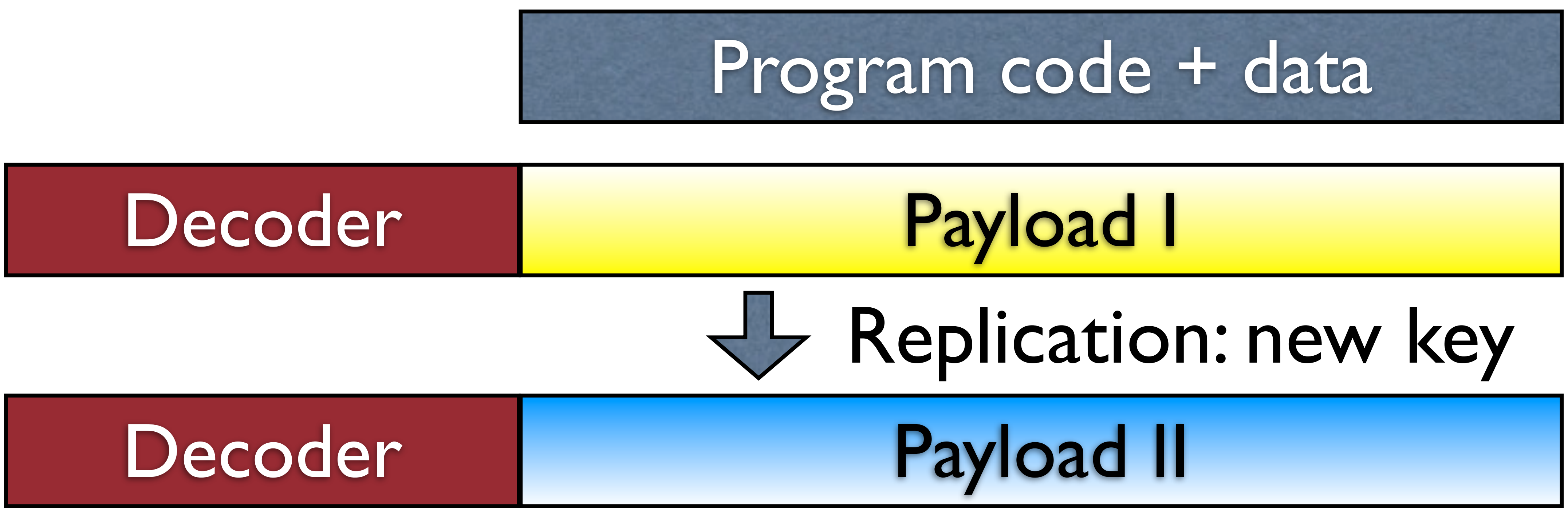
Polymorphism

- Program code and data are encoded
- Decoder prepares the program and then executes it
- Key is changed in each iteration



Polymorphism

- Program code and data are encoded
- Decoder prepares the program and then executes it
- Key is changed in each iteration



Example: Polymorphic Shellcode


```
.text:00401020 ; ---------------------------------------------------------------
.text:00401020                 mov     ecx, 0C7h
.text:00401025                 mov     dl, 13h
.text:00401027                 call    $+5
.text:0040102C                 pop     esi
.text:0040102D
.text:0040102D loc_40102D:                             ; CODE XREF: .text:00401031↓j
.text:0040102D                 xor     [ecx+esi+7], dl
.text:00401031                 loop    loc_40102D
.text:00401031 ; ---------------------------------------------------------------
.text:00401033                 db 0FDh ; ²
.text:00401034                 db 0EAh ; Û
.text:00401035                 db  8Ah ; è
.text:00401036                 db    4
.text:00401037                 db    5
.text:00401038                 db    6
.text:00401039                 db  43h ; C
.text:0040103A                 db  51h ; Q
.text:0040103B                 db 0ECh ; ý
.text:0040103C                 db  3Bh ; ;
.text:0040103D                 db 0D9h ; +
.text:0040103E                 db  44h ; D
.text:0040103F                 db  56h ; V
.text:00401040                 db  5Ch ; \
.text:00401041                 db  3Fh ; ?
.text:00401042                 db  9Bh ; ø
.text:00401043                 db  2Bh ; +
```

Note: The $ is used to refer to the same location where the instruction starts

Example: Polymorphic Shellcode

Note: The $ is used to refer to the same location where the instruction starts

Decrypter

```
.text:00401020 ; ---------------------------------------------------------------
.text:00401020                 mov     ecx, 0C7h
.text:00401025                 mov     dl, 13h
.text:00401027                 call    $+5
.text:0040102C                 pop     esi
.text:0040102D
.text:0040102D loc_40102D:                             ; CODE XREF: .text:00401031↓j
.text:0040102D                 xor     [ecx+esi+7], dl
.text:00401031                 loop    loc_40102D
.text:00401031 ; ---------------------------------------------------------------
```

Payload

```
.text:00401033                 db 0FDh ; ²
.text:00401034                 db 0EAh ; Û
.text:00401035                 db  8Ah ; è
.text:00401036                 db    4
.text:00401037                 db    5
.text:00401038                 db    6
.text:00401039                 db  43h ; C
.text:0040103A                 db  51h ; Q
.text:0040103B                 db 0ECh ; ý
.text:0040103C                 db  3Bh ; ;
.text:0040103D                 db 0D9h ; +
.text:0040103E                 db  44h ; D
.text:0040103F                 db  56h ; V
.text:00401040                 db  5Ch ; \
.text:00401041                 db  3Fh ; ?
.text:00401042                 db  9Bh ; ø
.text:00401043                 db  2Bh ; +
```

Example: Polymorphic Shellcode

Decrypter

Payload

```
.text:00401020 ; ---------------------------------------------------------------
.text:00401020                 mov     ecx, 0C7h
.text:00401025                 mov     dl, 13h
.text:00401027                 call    $+5
.text:0040102C                 pop     esi
.text:0040102D
.text:0040102D loc_40102D:                             ; CODE XREF: .text:00401031↓j
.text:0040102D                 xor     [ecx+esi+7], dl
.text:00401031                 loop    loc_40102D
.text:00401031 ; ---------------------------------------------------------------
.text:00401033                 db 0FDh ; ²
.text:00401034                 db 0EAh ; Û
.text:00401035                 db  8Ah ; è
.text:00401036                 db    4
.text:00401037                 db    5
.text:00401038                 db    6
.text:00401039                 db  43h ; C
.text:0040103A                 db  51h ; Q
.text:0040103B                 db 0ECh ; ý
.text:0040103C                 db  3Bh ; ;
.text:0040103D                 db 0D9h ; +
.text:0040103E                 db  44h ; D
.text:0040103F                 db  56h ; V
.text:00401040                 db  5Ch ; \
.text:00401041                 db  3Fh ; ?
.text:00401042                 db  9Bh ; ø
.text:00401043                 db  2Bh ; +
```

Note: The $ is used to refer to the same location where the instruction starts

Key is stored in `dl` and can be changed

Metamorphism

- Metamorphic code: the complete code is transformed, there are no constant parts
- Implementation is non-trivial
- Transformation must not change the semantics
- Example: change register assignment

```
mov eax, [401000h]
xor eax, [401004h]
mov [401000h], eax
```

→

```
mov ecx, [401000h]
xor ecx, [401004h]
mov [401000h], ecx
```

Self-modifying Code


```
.text:00401020                 mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                 add     ecx, edx        ; DATA XREF: .text:00401020↑w
.text:00401029                 retn
```

Self-modifying Code

- What does the following code do?

```
.text:00401020                 mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                 add     ecx, edx          ; DATA XREF: .text:00401020↑w
.text:00401029                 retn
```

Self-modifying Code

- What does the following code do?

```
.text:00401020                  mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                  add     ecx, edx        ; DATA XREF: .text:00401020↑w
.text:00401029                  retn
```

Self-modifying Code

- What does the following code do?

```
.text:00401020                 mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                 add     ecx, edx        ; DATA XREF: .text:00401020↑w
.text:00401029                 retn
```

Self-modifying Code

- What does the following code do?

```
.text:00401020                 mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                 add     ecx, edx        ; DATA XREF: .text:00401020↑w
.text:00401029                 retn
```

- First instruction *overwrites* parts of its own code, thereby *changing* the actual instruction

Self-modifying Code

- What does the following code do?

```
.text:00401020                 mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                 add     ecx, edx        ; DATA XREF: .text:00401020↑w
.text:00401029                 retn
```

- First instruction *overwrites* parts of its own code, thereby *changing* the actual instruction
- New instruction is: `add ecx, eax`

Self-modifying Code

Systems Security
Ruhr-University Bochum

- What does the following code do?

```
.text:00401020                 mov     byte ptr ds:loc_401027+1, 0C8h
.text:00401027                 add     ecx, edx        ; DATA XREF: .text:00401020↑w
.text:00401029                 retn
```

- First instruction *overwrites* parts of its own code, thereby *changing* the actual instruction
- New instruction is: `add ecx, eax`
- SMC is hard to analyze, disassembler needs to have some kind of heuristic to analyze such code

Unaligned Branches


```
:004000F0                 jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                             ; CODE XREF: start↑j
:004000F2                 imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                 imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                 imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start           endp ; sp-analysis failed
:00400108
:00400113                 imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                 db      64h
:0040011E                 popa
:00400120                 and     [eax], eax
:00400122
```

Unaligned Branches

```
:004000F0                 jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                             ; CODE XREF: start↑j
:004000F2                 imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                 imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                 imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start           endp ; sp-analysis failed
:00400108
:00400113                 imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                 db      64h
:0040011E                 popa
:00400120                 and     [eax], eax
:00400122
```

Unaligned Branches

```
:004000F0                  jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                                      ; CODE XREF: start↑j
:004000F2                  imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                  imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                  imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start            endp ; sp-analysis failed
:00400108
:00400113                  imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                  db      64h
:0040011E                  popa
:00400120                  and     [eax], eax
:00400122
```

Unaligned Branches

```
:004000F0                 jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                             ; CODE XREF: start↑j
:004000F2                 imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                 imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                 imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start           endp ; sp-analysis failed
:00400108
:00400113                 imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                 db      64h
:0040011E                 popa
:00400120                 and     [eax], eax
:00400122
```

Unaligned Branches

```
:004000F0                 jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                             ; CODE XREF: start↑j
:004000F2                 imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                 imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                 imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start           endp ; sp-analysis failed
:00400108
:00400113                 imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                 db      64h
:0040011E                 popa
:00400120                 and     [eax], eax
:00400122
```

- First jump target is the middle of another instruction

Unaligned Branches

```
:004000F0                 jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                             ; CODE XREF: start↑j
:004000F2                 imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                 imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                 imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start           endp ; sp-analysis failed
:00400108
:00400113                 imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                 db      64h
:0040011E                 popa
:00400120                 and     [eax], eax
:00400122
```

- First jump target is the middle of another instruction
- Thus another instruction is actually executed than displayed by the disassembler

Unaligned Branches

```
:004000F0                 jmp     short near ptr loc_4000F2+2
:004000F2 ; ---------------------------------------------------------------------------
:004000F2
:004000F2 loc_4000F2:                             ; CODE XREF: start↑j
:004000F2                 imul    eax, [edx+ebp*2+11C6840h], 2EB0040h
:004000FD                 imul    eax, dword ptr loc_400122[eax+ebp*2], 2EB9090h
:00400108                 imul    eax, [edx+ebp*2+1EE800h], 2EB0000h
:00400108 start           endp ; sp-analysis failed
:00400108
:00400113                 imul    eax, [edx+ebp*2+19E800h], 61540000h
:0040011E                 db      64h
:0040011E                 popa
:00400120                 and     [eax], eax
:00400122
```

- First jump target is the middle of another instruction
- Thus another instruction is actually executed than displayed by the disassembler
- The disassembler needs to take care of this aspect
  - *Linear sweep* vs. *recursive traversal*

Unaligned Branches

Correct disassembly

```
004000F0                jmp     short loc_4000F4
004000F0 ; ---------------------------------------------------------------------------
004000F2                db  69h ; i
004000F3                db  84h ; ä
004000F4 ; ---------------------------------------------------------------------------
004000F4
004000F4 loc_4000F4:                            ; CODE XREF: start↑j
004000F4                push    40h             ; uType
004000F6                push    offset Caption  ; "Tada!"
004000FB                jmp     short loc_4000FF
004000FB ; ---------------------------------------------------------------------------
004000FD                db  69h ; i
004000FE                db  84h ; ä
004000FF ; ---------------------------------------------------------------------------
004000FF
004000FF loc_4000FF:                            ; CODE XREF: start+B↑j
004000FF                push    offset Text     ; "Hello World!"
00400104                nop
00400105                nop
00400106                jmp     short loc_40010A
00400106 ; ---------------------------------------------------------------------------
00400108                db  69h ; i
00400109                db  84h ; ä
0040010A ; ---------------------------------------------------------------------------
0040010A
0040010A loc_40010A:                            ; CODE XREF: start+16↑j
0040010A                push    0               ; hWnd
0040010C                call    MessageBoxA
00400111                jmp     short loc_400115
```

Unaligned Branches

- Machinecode is *unaligned* for x86
  - No fixed instruction length
  - No boundary where code is aligned
- An attacker can abuse this
  - “Hide” instruction A within instruction B
  - Overlap two instructions
- *Return oriented programming* is based on this

Questions?

Systems Security
Ruhr-University Bochum

Contact:

Prof. Thorsten Holz

thorsten.holz@rub.de
@thorstenholz on Twitter

More information:

http://syssec.rub.de

http://moodle.rub.de

Sources

- Lecture *Software Reverse Engineering* at University of Mannheim, spring term 2010 (Ralf Hund, Carsten Willems and Felix Freiling)
- Rolf Rolles: "Binary Literacy", 2007
  - Highly recommended reading!
  - See link in Moodle