# cryptexec: runtime binary encryption using on-demand function extraction

©2004,2005 Željko Vrba

Please excuse my awkward English, it is not my native language.

What is binary encryption and why encrypt at all? For the answer to this question the reader is referred to the Phrack#58 [1] and article therein titled "Runtime binary encryption". This article describes a method to control the target program that doesn't does not rely on any assistance from the OS kernel or processor hardware. The method is implemented in x86-32 GNU AS (AT&T syntax). Once the controlling method is devised, it is relatively trivial to include on-the-fly code decryption.



## 1 Introduction

First let me introduce some terminology used in this article so that the reader is not confused.

- The attributes "target", "child" and "traced" are used interchangeably (depending on the context) to refer to the program being under the control of another program.
- The attributes "controlling" and "tracing" are used interchangeably to refer to the program that controls the target (debugger, strace, etc.)

## 2 OS- and hardware-assisted tracing

Current debuggers (both under Windows and UNIX) use x86 hardware features for debugging. The two most commonly used features are the trace flag (TF) and INT3 instruction, which has a convenient 1-byte encoding of 0xCC.

TF resides in bit 8 of the EFLAGS register and when set to 1 the processor generates exception#1 (debug exception) after each instruction is executed. When INT3 is executed, the processor generates exception#3 (breakpoint).

The traditional way to trace a program under UNIX is the `ptrace(2)` syscall. The program doing the trace usually does the following (shown in pseudocode):

```
fork()
  child:  ptrace(PT_TRACE_ME)
          execve("the program to trace")
  parent: controls the traced program with other ptrace() calls
```

Another way is to do `ptrace(PT_ATTACH)` on an already existing process. Other operations that ptrace() interface offers are reading/writing target instruction/data memory, reading/writing registers or continuing the execution (continually or up to the next system call – this capability is used by the well-known strace(1) program).

Each time the traced program receives a signal, the controlling program's `ptrace()` function returns. When the TF is turned on, the traced program receives a SIGTRAP after each instruction. The TF is usually not turned on by the traced program[1], but from the `ptrace(PT_STEP)`.

Unlike TF, the controlling program places 0xCC opcode at strategic[2] places in the code. The first byte of the instruction is replaced with 0xCC and the controlling program stores both the address and the original opcode. When execution comes to that address, SIGTRAP is delivered and the controlling program regains control. Then it replaces (again using ptrace()) 0xCC with original opcode and single-steps the original instruction. After that the original opcode is usually again replaced with 0xCC.

Although powerful, `ptrace()` has several disadvantages:

1. The traced program can be ptrace()d only by *one* controlling program.
2. The controlling and traced program live in separate address spaces, which makes changing traced memory awkward.
3. ptrace() is a system call: it is *slow* if used for full-blown tracing of larger chunks of code.

I won't go deeper in the mechanics of ptrace(), there are available tutorials [2] and the man page is pretty self-explanatory.

# 3 Userland tracing

The tracing can be done solely from the user-mode: the instructions are executed natively, *except control-transfer instructions* (CALL, JMP, Jcc, RET, LOOP, JCXZ). The background of this idea is explained nicely in [3] on the primitive 1960's MIX computer designed by Knuth.

Features of the method I'm about to describe:

- It allows that only portions of the executable file are encrypted.
- Different portions of the executable can be encrypted with different keys provided there is no cross-calling between them.

[1] Although nothing prevents it to do so - it is in the user-modifiable portion of EFLAGS.
[2] Usually the person doing the debugging decides what is strategic.

- It allows encrypted code to freely call non-encrypted code. In this case the non-encrypted code is also executed instruction by instruction. When called outside of encrypted code, it still executes without tracing.
- There is never more than 24 bytes of encrypted code held in memory in plaintext.
- OS- and language-independent.

The rest of this section explains the provided API, gives a high-level description of the implementation, shows a usage example and discusses Here are the details of my own implementation.

## 3.1 Provided API

No "official" header file is provided. Because of the sloppy and convenient C parameter passing and implicit function declarations, you can get away with no declarations whatsoever.

The decryption API consists of one typedef and one function. The following is the generic prototype that your decryption routine must fit:

```
typedef (*decrypt_fn_ptr)(void *key, unsigned char *dst, const unsigned
  char *src);
```

It is called from the main decryption routine with the following arguments:

**key** is a pointer to decryption key data. Note that in most cases this is NOT the raw key but pointer to some kind of "decryption context".

**dst** is a pointer to destination buffer.

**src** is a pointer to source buffer.

Note that there is no size argument: the block size is fixed to 8 bytes. The routine should not read more than 8 bytes from the src and *never* output more than 8 bytes to dst.

Another unusual constraint is that the decryption function *must not modify its arguments on the stack*. If you need to do this, copy the stack arguments into local variables. This is a consequence of how the routine is called from within the decryption engine – see the code for details.

There are no constraints whatsoever on the kind of encryption which can be used. *Any bijective function* which maps 8 bytes to 8 bytes is suitable. Encrypt the code with the function, and use its inverse for the decryption. If you use the identity function, then decryption becomes simple single-stepping with no hardware support – see section 4 for related work.

The entry point to the decryption engine is the following function:

```
int crypt_exec(decrypt_fn_ptr dfn, const void *key, const void *lo_addr,
  const void *hi_addr, const void *F, ...);
```

The decryption function has the capability to switch between executing both encrypted and plaintext code. The encrypted code can call the plain-text code and plain-text code can return into the encrypted code. But for that to be possible, it needs to know the address bounds of the encrypted code.

Note that this function *is not reentrant!* It is not allowed for *any* kind of code (either plaintext or encrypted) running under the `crypt_exec` routine to call `crypt_exec` again. Things will **break badly** because the internal state of previous invocation is statically allocated and will get overwritten.

The arguments are as follows:

**dfn** points to decryption function. The function is called with the key argument provided to `crypt_exec` and the addresses of destination and source buffers.

**key** This usually doesn't point to the raw key bytes, but the initialized decryption context. See the example code for the test2 program: first the user-provided raw key is loaded into the decryption context and the address of the *context* is given to the `crypt_exec` function.

**lo_addr, hi_addr** are low and high addresses that are encrypted under the same key. This is to facilitate calling non-encrypted code from within encrypted code.

**F** points to the code which should be executed under the decryption engine. It can be an ordinary C function pointer. Since the tracing routine[3] was written with 8-byte block ciphers in mind, the F function must be at least 8-byte aligned and its length must be a multiple of 8. This is easier to achieve (even with standard C) than it sounds. See the example below.

**...** become arguments to the called function.

`crypt_exec` arranges function F to be called with the arguments provided in the varargs list. When `crypt_exec` returns, its return value is what the F returned. In short, the call

```
x = crypt_exec(dfn, key, lo_addr, hi_addr, F, ...);
```

has *exactly the same semantics* as

```
x = F(...);
```

would have, were F plain-text.

Currently, the code is tailored to use the XDE disassembler. Other disassemblers can be used, but the code which accesses results must be changed in few places (all references to the disbuf variable).

The `crypt_exec` routine provides a private stack of 4kB. If you use your own decryption routine and/or disassembler, take care not to consume too much stack space. If you want to enlarge the local stack, look for the `local_stk` label in the code.

## 3.2 High-level description

The tracing routine maintains two contexts: the traced context and its own context. The context consists of 8 32-bit general-purpose registers and flags. Other registers are not modified by the routine. Both contexts are held on the private stack (that is also used for calling C).

The idea is to fetch, one at a time, instructions from the traced program and execute them natively. Intel instruction set has rather irregular encoding, so the XDE [5] disassembler engine is used to find both the real opcode and total instruction length. During experiments on FreeBSD (which uses LOCK- prefixed MOV instruction in its dynamic loader) I discovered a bug in XDE which is described and fixed below.

[3] In the rest of this article I will call this interchangeably tracing or decryption routine. In fact, this is a tracing routine with added decryption.

We maintain our own EIP in `traced_eip`, round it down to the next lower 8-byte boundary and then decrypt[4] 24 bytes[5] into our own buffer. Then the disassembly takes place and the control is transferred to emulation routines via the opcode control table. All instructions, except control transfer, are executed natively (in traced context which is restored at appropriate time). After single instruction execution, the control is returned to our tracing routine.

In order to prevent losing control, the control transfer instructions[6] are *emulated*. The big problem was (until I solved it) emulating indirect JMP and CALL instructions (which can appear with any kind of complex EA that i386 supports). The problem is solved by replacing the CALL/JMP instruction with MOV to register opcode, and modifying bits 3-5 (reg field) of modR/M byte to set the target register (this field holds the part of opcode in the CALL/JMP case). Then we let the processor to calculate the EA for us.

Of course, a means are needed to stop the encrypted execution and to enable encrypted code to call plaintext code:

1. On entering, the tracing engine pops the return addreess and its private arguments and then pushes the return address back to the traced stack. At that moment:
   - The stack frame is good for executing a regular C function (F).
   - The top of stack pointer (esp) is stored into `end_esp`.
2. When the tracing routine encounters a RET instruction it first checks the `traced_esp`. If it equals `end_esp`, it is a point where the F function would have ended. Therefore, we restore the traced context and *do not emulate* RET, but let it execute natively. This way the tracing routine loses control and normal instruction execution continues.

In order to allow encrypted code to call plaintext code, there are `lo_addr` and `hi_addr` parameters. These parameters determine the low and high boundary of encrypted code in memory. If the `traced_eip` falls out of `[lo_addr, hi_addr)` range, the decryption routine pointer is swapped with the pointer to a no-op "decryption" that just copies 8 bytes from source to destination. When the `traced_eip` again falls into that interval, the pointers are again swapped.

## 3.3 Actual usage example

Given encrypted execution engine, how do we test it? For this purpose I have written a small utility named cryptfile that encrypts a portion of the executable file (`$` is UNIX prompt):

```
$ gcc -c cast5.c
$ gcc cryptfile.c cast5.o -o cryptfile
$ ./cryptfile
USAGE: ./cryptfile <-e|-d> FILE KEY STARTOFF ENDOFF
KEY MUST be 32 hex digits (128 bits).
```

The parameters are as follows:

**-e,-d** one of these is MANDATORY and stands for encryption or decryption.

[4] The decryption routine is called indirectly for reasons described later.

[5] The number comes from worst-case considerations: if an instruction begins at a boundary that is $\equiv 7 \pmod 8$, given maximum instruction length of 15 bytes, yields a total of 22 bytes = 3 blocks. The buffer has 32 bytes in order to accomodate an additional JMP indirect instruction after the traced instruction. The JMP jumps *indirectly* to place in the tracing routine where execution should continue.

[6] INT instructions are *not* considered as control transfer. After (if) the OS returns from the invoked trap, the program execution continues sequentially, the instruction right after INT. So there are no special measures that should be taken.

**FILE** is the executable file to be encrypted.

**KEY** is the encryption key. It must be given as 32 hex digits.

**STARTOFF, ENDOFF** are the starting and ending offset in the file that should be encrypted.

They *must* be a multiple of block size (8 bytes). If not, the file will be correctly encrypted, but the encrypted execution will not work correctly.

The whole package is tested on a simple program, test2.c. This program demonstrates that encrypted functions can call both encrypted and plaintext functions as well as return results. It also demonstrates that the engine works even when calling functions in shared libraries.

Now we build the encrypted execution engine:

```
$ gcc -c crypt_exec.S
$ cd xde101
$ gcc -c xde.c
$ cd ..
$ ld -r cast5.o crypt_exec.o xde101/xde.o -o crypt_monitor.o
```

I'm using patched XDE. The last step is to combine several relocatable object files in a single relocatable file for easier linking with other programs.

Then we proceed to build the test program. We must ensure that functions that we want to encrypt are aligned to 8 bytes. I'm specifying 16, just in case. Therefore:

```
$ gcc -falign-functions=16 -g test2.c crypt_monitor.o -o test2
```

We want to encrypt functions `f1` and `f2`. How do we map from function names to offsets in the executable file? Fortunately, this can be simply done for ELF with the readelf utility (that's why I chose such an awkward way – I didn't want to bother with yet another ELF "parser").

```
$ readelf -s test2

Symbol table '.dynsym' contains 23 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048484    57 FUNC    GLOBAL DEFAULT  UND printf
     2: 08050aa4     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
     3: 08048494     0 FUNC    GLOBAL DEFAULT  UND memcpy
     4: 08050b98     4 OBJECT  GLOBAL DEFAULT  20 __stderrp
     5: 08048468     0 FUNC    GLOBAL DEFAULT   8 _init
     6: 08051c74     4 OBJECT  GLOBAL DEFAULT  20 environ
     7: 080484a4    52 FUNC    GLOBAL DEFAULT  UND fprintf
     8: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __deregister_frame_info
     9: 0804fc00     4 OBJECT  GLOBAL DEFAULT  13 __progname
    10: 080484b4   172 FUNC    GLOBAL DEFAULT  UND sscanf
    11: 08050b98     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    12: 080484c4     0 FUNC    GLOBAL DEFAULT  UND memset
    13: 0804ca64     0 FUNC    GLOBAL DEFAULT  11 _fini
    14: 080484d4   337 FUNC    GLOBAL DEFAULT  UND atexit
    15: 080484e4   121 FUNC    GLOBAL DEFAULT  UND scanf
    16: 08050b98     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
    17: 08050b68     0 OBJECT  GLOBAL DEFAULT  ABS _GLOBAL_OFFSET_TABLE_
    18: 08051c78     0 NOTYPE  GLOBAL DEFAULT  ABS _end
    19: 080484f4   101 FUNC    GLOBAL DEFAULT  UND exit
    20: 08048504     0 FUNC    GLOBAL DEFAULT  UND strlen
    21: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    22: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __register_frame_info
```

```
Symbol table '.symtab' contains 145 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 080480f4     0 SECTION LOCAL  DEFAULT    1
     2: 08048110     0 SECTION LOCAL  DEFAULT    2
     3: 08048128     0 SECTION LOCAL  DEFAULT    3
     4: 080481d0     0 SECTION LOCAL  DEFAULT    4
     5: 08048340     0 SECTION LOCAL  DEFAULT    5
     6: 08048418     0 SECTION LOCAL  DEFAULT    6
     7: 08048420     0 SECTION LOCAL  DEFAULT    7
     8: 08048468     0 SECTION LOCAL  DEFAULT    8
     9: 08048474     0 SECTION LOCAL  DEFAULT    9
    10: 08048520     0 SECTION LOCAL  DEFAULT   10
    11: 0804ca64     0 SECTION LOCAL  DEFAULT   11
    12: 0804ca80     0 SECTION LOCAL  DEFAULT   12
    13: 0804fc00     0 SECTION LOCAL  DEFAULT   13
    14: 08050aa0     0 SECTION LOCAL  DEFAULT   14
    15: 08050aa4     0 SECTION LOCAL  DEFAULT   15
    16: 08050b54     0 SECTION LOCAL  DEFAULT   16
    17: 08050b5c     0 SECTION LOCAL  DEFAULT   17
    18: 08050b64     0 SECTION LOCAL  DEFAULT   18
    19: 08050b68     0 SECTION LOCAL  DEFAULT   19
    20: 08050b98     0 SECTION LOCAL  DEFAULT   20
    21: 00000000     0 SECTION LOCAL  DEFAULT   21
    22: 00000000     0 SECTION LOCAL  DEFAULT   22
    23: 00000000     0 SECTION LOCAL  DEFAULT   23
    24: 00000000     0 SECTION LOCAL  DEFAULT   24
    25: 00000000     0 SECTION LOCAL  DEFAULT   25
    26: 00000000     0 SECTION LOCAL  DEFAULT   26
    27: 00000000     0 SECTION LOCAL  DEFAULT   27
    28: 00000000     0 SECTION LOCAL  DEFAULT   28
    29: 00000000     0 SECTION LOCAL  DEFAULT   29
    30: 00000000     0 SECTION LOCAL  DEFAULT   30
    31: 00000000     0 SECTION LOCAL  DEFAULT   31
    32: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    33: 08050b54     0 OBJECT  LOCAL  DEFAULT   16 __CTOR_LIST__
    34: 08050b5c     0 OBJECT  LOCAL  DEFAULT   17 __DTOR_LIST__
    35: 08050aa0     0 OBJECT  LOCAL  DEFAULT   14 __EH_FRAME_BEGIN__
    36: 08050b64     0 OBJECT  LOCAL  DEFAULT   18 __JCR_LIST__
    37: 0804fc08     0 OBJECT  LOCAL  DEFAULT   13 p.0
    38: 08050b9c     1 OBJECT  LOCAL  DEFAULT   20 completed.1
    39: 080485b0     0 FUNC    LOCAL  DEFAULT   10 __do_global_dtors_aux
    40: 08050ba0    24 OBJECT  LOCAL  DEFAULT   20 object.2
    41: 08048610     0 FUNC    LOCAL  DEFAULT   10 frame_dummy
    42: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    43: 08050b58     0 OBJECT  LOCAL  DEFAULT   16 __CTOR_END__
    44: 08050b60     0 OBJECT  LOCAL  DEFAULT   17 __DTOR_END__
    45: 08050aa0     0 OBJECT  LOCAL  DEFAULT   14 __FRAME_END__
    46: 08050b64     0 OBJECT  LOCAL  DEFAULT   18 __JCR_END__
    47: 0804ca30     0 FUNC    LOCAL  DEFAULT   10 __do_global_ctors_aux
    48: 00000000     0 FILE    LOCAL  DEFAULT  ABS test2.c
    49: 08048660    75 FUNC    LOCAL  DEFAULT   10 f1
    50: 080486b0    58 FUNC    LOCAL  DEFAULT   10 f2
    51: 08050bb8    16 OBJECT  LOCAL  DEFAULT   20 key.0
    52: 080486f0   197 FUNC    LOCAL  DEFAULT   10 decode_hex_key
    53: 00000000     0 FILE    LOCAL  DEFAULT  ABS cast5.c
    54: 0804cba0  1024 OBJECT  LOCAL  DEFAULT   12 s1
    55: 0804cfa0  1024 OBJECT  LOCAL  DEFAULT   12 s2
    56: 0804d3a0  1024 OBJECT  LOCAL  DEFAULT   12 s3
    57: 0804d7a0  1024 OBJECT  LOCAL  DEFAULT   12 s4
    58: 0804dba0  1024 OBJECT  LOCAL  DEFAULT   12 s5
```

```
 59: 0804dfa0  1024 OBJECT  LOCAL  DEFAULT  12 s6
 60: 0804e3a0  1024 OBJECT  LOCAL  DEFAULT  12 s7
 61: 0804e7a0  1024 OBJECT  LOCAL  DEFAULT  12 sb8
 62: 0804a3c0  3734 FUNC    LOCAL  DEFAULT  10 key_schedule
 63: 0804b408     0 NOTYPE  LOCAL  DEFAULT  10 identity_decrypt
 64: 08051bf0     0 NOTYPE  LOCAL  DEFAULT  20 r_decrypt
 65: 08051be8     0 NOTYPE  LOCAL  DEFAULT  20 key
 66: 08050bd4     0 NOTYPE  LOCAL  DEFAULT  20 lo_addr
 67: 08050bd8     0 NOTYPE  LOCAL  DEFAULT  20 hi_addr
 68: 08050bcc     0 NOTYPE  LOCAL  DEFAULT  20 traced_eip
 69: 08050be0     0 NOTYPE  LOCAL  DEFAULT  20 end_esp
 70: 08050bd0     0 NOTYPE  LOCAL  DEFAULT  20 traced_ctr
 71: 0804b449     0 NOTYPE  LOCAL  DEFAULT  10 decryptloop
 72: 08050bc8     0 NOTYPE  LOCAL  DEFAULT  20 traced_esp
 73: 08051be4     0 NOTYPE  LOCAL  DEFAULT  20 stk_end
 74: 0804b456     0 NOTYPE  LOCAL  DEFAULT  10 decryptloop_nocontext
 75: 0804b476     0 NOTYPE  LOCAL  DEFAULT  10 .store_decrypt_ptr
 76: 08051bec     0 NOTYPE  LOCAL  DEFAULT  20 decrypt
 77: 0804fc35     0 NOTYPE  LOCAL  DEFAULT  13 insn
 78: 08051bf4     0 NOTYPE  LOCAL  DEFAULT  20 disbuf
 79: 08051be4     0 NOTYPE  LOCAL  DEFAULT  20 ilen
 80: 080501f0     0 NOTYPE  LOCAL  DEFAULT  13 continue
 81: 0804fdf0     0 NOTYPE  LOCAL  DEFAULT  13 control_table
 82: 0804fc20     0 NOTYPE  LOCAL  DEFAULT  13 _unhandled
 83: 0804fc21     0 NOTYPE  LOCAL  DEFAULT  13 _nonjump
 84: 0804fc33     0 NOTYPE  LOCAL  DEFAULT  13 .execute
 85: 0804fc55     0 NOTYPE  LOCAL  DEFAULT  13 _jcc_rel8
 86: 0804fc5e     0 NOTYPE  LOCAL  DEFAULT  13 _jcc_rel32
 87: 0804fc65     0 NOTYPE  LOCAL  DEFAULT  13 ._jcc_rel32_insn
 88: 0804fc71     0 NOTYPE  LOCAL  DEFAULT  13 ._jcc_rel32_true
 89: 0804fc6b     0 NOTYPE  LOCAL  DEFAULT  13 ._jcc_rel32_false
 90: 0804fc72     0 NOTYPE  LOCAL  DEFAULT  13 rel_offset_fixup
 91: 0804fc7d     0 NOTYPE  LOCAL  DEFAULT  13 _retn
 92: 0804fca6     0 NOTYPE  LOCAL  DEFAULT  13 ._endtrace
 93: 0804fcbe     0 NOTYPE  LOCAL  DEFAULT  13 _loopne
 94: 0804fce0     0 NOTYPE  LOCAL  DEFAULT  13 ._loop_insn
 95: 0804fcd7     0 NOTYPE  LOCAL  DEFAULT  13 ._doloop
 96: 0804fcc7     0 NOTYPE  LOCAL  DEFAULT  13 _loope
 97: 0804fcd0     0 NOTYPE  LOCAL  DEFAULT  13 _loop
 98: 0804fcec     0 NOTYPE  LOCAL  DEFAULT  13 ._loop_insn_true
 99: 0804fce2     0 NOTYPE  LOCAL  DEFAULT  13 ._loop_insn_false
100: 0804fcf6     0 NOTYPE  LOCAL  DEFAULT  13 _jcxz
101: 0804fd0a     0 NOTYPE  LOCAL  DEFAULT  13 _callrel
102: 0804fd0f     0 NOTYPE  LOCAL  DEFAULT  13 _call
103: 0804fd38     0 NOTYPE  LOCAL  DEFAULT  13 _jmp_rel8
104: 0804fd41     0 NOTYPE  LOCAL  DEFAULT  13 _jmp_rel32
105: 0804fd49     0 NOTYPE  LOCAL  DEFAULT  13 _grp5
106: 0804fda4     0 NOTYPE  LOCAL  DEFAULT  13 ._grp5_continue
107: 08050bdc     0 NOTYPE  LOCAL  DEFAULT  20 our_esp
108: 0804fdc9     0 NOTYPE  LOCAL  DEFAULT  13 ._grp5_call
109: 0804fdd0     0 NOTYPE  LOCAL  DEFAULT  13 _0xf
110: 08050be4     0 NOTYPE  LOCAL  DEFAULT  20 local_stk
111: 00000000     0 FILE    LOCAL  DEFAULT  ABS xde.c
112: 0804b419     0 NOTYPE  GLOBAL DEFAULT  10 crypt_exec
113: 08048484    57 FUNC    GLOBAL DEFAULT  UND printf
114: 08050aa4     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
115: 08048494     0 FUNC    GLOBAL DEFAULT  UND memcpy
116: 0804b684  4662 FUNC    GLOBAL DEFAULT  10 xde_disasm
117: 08050b98     4 OBJECT  GLOBAL DEFAULT  20 __stderrp
118: 0804fc04     0 OBJECT  GLOBAL HIDDEN   13 __dso_handle
119: 0804b504   384 FUNC    GLOBAL DEFAULT  10 reg2xset
```

```
   120: 08048468     0 FUNC    GLOBAL DEFAULT   8 _init
   121: 0804c8bc   364 FUNC    GLOBAL DEFAULT  10 xde_asm
   122: 08051c74     4 OBJECT  GLOBAL DEFAULT  20 environ
   123: 080484a4    52 FUNC    GLOBAL DEFAULT  UND fprintf
   124: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __deregister_frame_info
   125: 0804fc00     4 OBJECT  GLOBAL DEFAULT  13 __progname
   126: 08048520   141 FUNC    GLOBAL DEFAULT  10 _start
   127: 0804b258   431 FUNC    GLOBAL DEFAULT  10 cast5_setkey
   128: 080484b4   172 FUNC    GLOBAL DEFAULT  UND sscanf
   129: 08050b98     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
   130: 080484c4     0 FUNC    GLOBAL DEFAULT  UND memset
   131: 080487c0   318 FUNC    GLOBAL DEFAULT  10 main
   132: 0804ca64     0 FUNC    GLOBAL DEFAULT  11 _fini
   133: 080484d4   337 FUNC    GLOBAL DEFAULT  UND atexit
   134: 080484e4   121 FUNC    GLOBAL DEFAULT  UND scanf
   135: 08050200  2208 OBJECT  GLOBAL DEFAULT  13 xde_table
   136: 08050b98     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
   137: 08050b68     0 OBJECT  GLOBAL DEFAULT  ABS _GLOBAL_OFFSET_TABLE_
   138: 08051c78     0 NOTYPE  GLOBAL DEFAULT  ABS _end
   139: 08049660  3421 FUNC    GLOBAL DEFAULT  10 cast5_decrypt
   140: 080484f4   101 FUNC    GLOBAL DEFAULT  UND exit
   141: 08048900  3421 FUNC    GLOBAL DEFAULT  10 cast5_encrypt
   142: 08048504     0 FUNC    GLOBAL DEFAULT  UND strlen
   143: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
   144: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __register_frame_info
```

We see that function `f1` has address 0x8048660 and size 75 = 0x4B. Function `f2` has address 0x80486B0 and size 58 = 3A. Simple calculation shows that they are in fact consecutive in memory so we don't have to encrypt them separately but in a single block ranging from 0x8048660 to 0x80486F0.

```
$ readelf -l test2

Elf file type is EXEC (Executable file)
Entry point 0x8048520
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00019 0x00019 R   0x1
      [Requesting program interpreter: /usr/libexec/ld-elf.so.1]
  LOAD           0x000000 0x08048000 0x08048000 0x06bed 0x06bed R E 0x1000
  LOAD           0x006c00 0x0804fc00 0x0804fc00 0x00f98 0x02078 RW  0x1000
  DYNAMIC        0x007aa4 0x08050aa4 0x08050aa4 0x000b0 0x000b0 RW  0x4
  NOTE           0x000110 0x08048110 0x08048110 0x00018 0x00018 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .rel.dyn .rel.plt
          .init .plt .text .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
   04     .dynamic
   05     .note.ABI-tag
```

From this we see that both addresses (0x8048660 and 0x80486F0) fall into the first LOAD segment which is loaded at VirtAddr 0x804800 and is placed at offset 0 in the file. Therefore, to map virtual

address to file offset we simply subtract 0x8048000 from each address giving 0x660 = 1632 and 0x6F0 = 1776.

If you obtain ELFsh [7] then you can make your life much easier. The following transcript shows how ELFsh can be used to obtain the same information:

```
$ elfsh

         Welcome to The ELF shell 0.51b3 .::.

         .::. This software is under the General Public License
         .::. Please visit http://www.gnu.org to know about Free Software

[ELFsh-0.51b3]$ load test2

 [*] New object test2 loaded on Mon Jun 13 20:45:33 2005

[ELFsh-0.51b3]$ sym f1

 [SYMBOL TABLE]
 [Object test2]

 [059] 0x8048680  FUNCTION f1
 size:0000000075 foffset:001632 scope:Local  sctndx:10 => .text + 304

[ELFsh-0.51b3]$ sym f2

 [SYMBOL TABLE]
 [Object test2]

 [060] 0x80486d0  FUNCTION f2
 size:0000000058 foffset:001776 scope:Local  sctndx:10 => .text + 384

[ELFsh-0.51b3]$ exit

 [*] Unloading object 1 (test2) *

         Good bye ! .::. The ELF shell 0.51b3
```

The field `foffset` gives the symbol offset within the executable, while `size` is its size. Here all the numbers are decimal.

Now we are ready to encrypt a part of the executable with a very 'imaginative' password and then test the program:

```
$ echo -n "password" | openssl md5
5f4dcc3b5aa765d61d8327deb882cf99
$ ./cryptfile -e test2 5f4dcc3b5aa765d61d8327deb882cf99 1632 1776
$ chmod +x test2.crypt
$ ./test2.crypt
```

At the prompt enter the same hex string and then enter numbers 12 and 34 for a and b. The result *must* be 1662, and esp before and after must be the same.

Once you are sure that the program works correctly, you can `strip(1)` symbols from it.

## 3.4 XDE bug

During the development, a I have found a bug in the XDE disassembler engine: it didn't correctly handle the LOCK (0xF0) prefix. Because of the bug XDE claimed that 0xF0 is a single-byte instruction. This is the needed patch to correct the disassembler:

```
--- xde.c      Sun Apr 11 02:52:30 2004
+++ xde_new.c  Mon Aug 23 08:49:00 2004
@@ -101,6 +101,8 @@
     if (c == 0xF0)
     {
       if (diza->p_lock != 0) flag |= C_BAD;     /* twice */
+      diza->p_lock = c;
+      continue;
     }

     break;
```

I also needed to remove `__cdecl` on functions, a 'feature' of Win32 C compilers not needed on UNIX platforms.

## 3.5 Limitations

- XDE engine (probably) can't handle new instructions (SSE, MMX, etc.). For certain it can't handle 3dNow! because they begin with 0x0F 0x0F, a byte sequence for which the XDE claims is an invalid instruction encoding.
- The tracer shares the same memory with the traced program. If the traced program is so badly broken that it writes to (random) memory it doesn't own, it can stumble upon and overwrite portions of the tracing routine.
- Each form of tracing has its own speed impacts. I didn't meausre how much this method slows down program execution (especially compared to `ptrace()`).
- Doesn't handle even all 386 instructions (most notably far calls/jumps and RET imm16). In this case the tracer stops with HLT which should cause GPF under any OS that runs user processes in rings other than 0.
- The block size of 8 bytes is hardcoded in many places in the program. The source (both C and ASM) should be parametrized by some kind of BLOCKSIZE `#define`.
- The tracing routine is *not reentrant*! Meaning, encrypted code can't call again crypt_exec because it will overwrite its own context!
- The code itself isn't optimal:
  - `identity_decrypt` could use 4-byte moves.
  - More registers could be used to minimize memory references.

## 3.6 Porting considerations

This is as heavy as it gets – there isn't a single piece of machine-independent code in the main routine that could be used on an another processor architecture. I believe that porting shouldn't be too difficult, mostly rewriting the mechanics of the current program. Some points to watch out for include:

- Be sure to handle all control flow instructions.
- Move instructions could affect processor flags.
- Write a disassembly routine. Most RISC architectures have regular instruction set and should be far easier to disassemble than x86 code.
- This is self-modifying code: flushing the instruction prefetch queue might be needed.
- Handle delayed jumps and loads if the architecture provides them. This could be tricky.
- You might need to get around page protections before calling the decryptor (non-executable data segments).

Due to unavailability of non-x86 hardware I wasn't able to implement the decryptor on another processor.

# 4 Related work

This section gives a brief overview of existing work, either because of similarity in coding techniques (ELFsh and tracing without `ptrace(2)`) or because of the code protection aspect.

## 4.1 ELFsh

The ELFsh crew's article on elfsh and e2dbg [7], also in this Phrack issue. A common point in our work is the approach to program tracing without using `ptrace(2)`. Their latest work is a scriptable embedded ELF debugger, e2dbg. They are also getting around PaX protections, an issue I didn't even take into account.

## 4.2 Shiva

The Shiva binary encryptor [8], released in binary-only form. It tries really hard to prevent reverse engineering by including features such as trap flag detection, `ptrace(2)` defense, demand-mapped blocks (so that fully decrpyted image can't be dumped via `/proc`), using INT3 to emulate some instructions, and by encryption in layers. The 2nd, password protected layer, is optional and encrypted using 128-bit AES. Layer 3 encryption uses TEA, the tiny encryption algorithm.

According to the analysis in [9], "for sufficiently large programs, no more than 1/3 of the program will be decrypted at any given time". This is MUCH larger amount of decrypted program text than in my case: 24 bytes, independent of any external factors. Also, Shiva is heavily tied to the ELF format, while my method is not tied to any operating system or executable format (although the current code IS limited to the 32-bit x86 architecture).

## 4.3 Burneye

There are actually two tools released by team-teso: burneye and burneye2 (objobf) [10].

Burneye is a powerful binary encryption tool. Similarly to Shiva, it has three layers: 1) obfuscation, 2) password-based encryption using RC4 and SHA1 (for generating the key from passphrase), and 3) the *fingerprinting layer*.

The fingerprinting layer is the most interesting one: the data about the target system is collected (e.g. amount of memory, etc..) and made into a "fingeprint". The executable is encrypted taking the fingerprint into account so that the resulting binary can be run only on the host with the given fingerprint. There are two fingerprinting options:

- Fingeprint tolerance can be specified so that Small deviations are allowed. That way, for example, the memory can be upgraded on the target system and the executable will still work. If the number of differences in the fingeprint is too large, the program won't work.
- Seal: the program produced with this option will run on any system. However, the first time it is run, it creats a fingerprint of the host and 'seals' itself to that host. The original seal binary is securely deleted afterwards.

The encrypted binary can also be made to delete itself when a certain environment variable is set during the program execution.

objobf is just relocatable object obfuscator. There is no encryption layer. The input is an ordinary relocatable object and the output is transformed, obfuscated, and functionally equivalent code. Code transformations include: inserting junk instructions, randomizing the order of basic blocks, and splitting basic blocks at random points.

# 5 Conclusion and future work

Highlights of the distinguishing features of the code encryption technique presented here:

- Very small amount of plaintext code in memory at any time - only 24 bytes. Other tools leave much more plain-text code in memory.
- No special loaders or executable format manipulations are needed. There is one simple utility that encrypts the existing code in-place. It is executable format-independent since its arguments are function offsets within the executable (which map to function addresses in runtime).
- The code is tied to the 32-bit x86 architecture, however it should be portable without changes to any operating system running on x86-32. Special arrangements for setting up page protections may be necessary if PaX or NX is in effect.

On the downside, the current version of the engine is very vulnerable with respect to reverse-engineering. It can be easily recognized by scanning for fixed sequences of instructions (the decryption routine). Once the decryptor is located, it is easy to monitor a few fixed memory addresses to obtain both the EIP and the original instruction residing at that EIP. The key material data is easy to obtain, but this is the case in *any* approach using in-memory keys.

However, the decryptor in its current form has one advantage: since it is ordinary code that does no special tricks, it should be easy to combine it with a tool that is more resilient to reverse-engineering, like Shiva or Burneye.

Following items highlight possible extensions to the method described in this article:

- Better encryption scheme. ECB mode is bad, especially with small block size of 8 bytes. Possible alternative is the following:
  1. Round the `traced_eip` down to a multiple of 8 bytes.
  2. Encrypt the result with the key.
  3. Xor the result with the instruction bytes.

  That way the encryption depends on the location in memory. Decryption works the same way. However, it would complicate cryptfile.c program.
- Encrypted data. Devise a transparent (for the C programmer) way to access the encrypted data. At least two approaches come to mind: 1) playing with page mappings and handling read/write faults, or 2) use XDE to decode all accesses to memory and perform encryption or decryption, depending on the type of access (read or write). The first approach seems too slow (many context switches per data access) to be practical.
- New instruction sets and architectures. Expand XDE to handle new x86 instructions. Port the routine to architectures other than i386 (first comes to mind AMD64, then ARM, SPARC...).
- Perform decryption on the *smart card*. This is slow, but there is no danger of key compromise.
- Polymorphic decryption engine.

# 6 Credits

Thanks go to mayhem who has reviewed this article. His suggestions were very helpful, making the text much more mature than the original.

# 7 References

1. Phrack magazine.
   http://www.phrack.org
2. ptrace tutorials:
   http://linuxgazette.net/issue81/sandeep.html
   http://linuxgazette.net/issue83/sandeep.html
   http://linuxgazette.net/issue85/sandeep.html

3. D. E. Knuth: The Art of Computer Programming, vol.1: Fundamental Algorithms.
4. Fenris.
   http://lcamtuf.coredump.cx/fenris/whatis.shtml
5. XDE.
   http://z0mbie.host.sk
6. Source code for described programs. The source I have written is released under MIT license. Other files have different licenses. The archive also contains a patched version of XDE.
   http://www.core-dump.com.hr/software/cryptexec.tar.gz
7. ELFsh, the ELF shell. A powerful program for manipulating ELF files.
   http://elfsh.devhell.org
8. Shiva binary encryptor.
   http://www.securereality.com.au
9. Reverse Engineering Shiva.
   http://blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf
10. Burneye and Burneye2 (objobf).
    http://packetstormsecurity.org/groups/teso/indexsize.html

# 8 The tracer source: crypt_exec.S

```
/*
Copyright (c) 2004 Zeljko Vrba

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to permit persons to
whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
```

```
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

.text
/*****************************************************************************
 * void *crypt_exec(
 *   decrypt_fn_ptr dfn, const void *key,
 *   const void *lo_addr, const void *hi_addr,
 *   const void *addr, ...)
 * typedef (*decrypt_fn_ptr)(
 *   void *key, unsigned char *dst, const unsigned char *src);
 *
 * - dfn is pointer to deccryption function
 * - key is pointer to crypt routine key data
 * - addr is the addres where execution should begin. due to the way the code
 *   is decrypted and executed, it MUST be aligned to 8 (BLOCKSIZE) bytes!!
 * - the rest are arguments to called function
 *
 * The crypt_exec stops when the stack pointer becomes equal to what it was
 * on entry, and executing 'ret' would cause the called function to exit. This
 * works assuming normal C compiled code.
 *
 * Returns the value the function would normally return.
 *
 * This code calls:
 * int xde_disasm(unsigned char *ip, struct xde_instr *outbuf);
 * XDE disassembler engine is compiled and used with PACKED structure!
 *
 * It is assumed that the encryption algorithm uses 64-bit block size. Very
 * good protection could be done if decryption is executed on the SMART CARD.
 *
 * Some terminology:
 * 'Traced' refers to the original program being executed instruction by
 * instruction. The technique used resembles Knuth's tracing routine (and
 * indeed, we get true tracing when decryption is dropped).
 *
 * 'Our' refers to our data stack, etc.
 *
 * TODOs and limitations:
 * - some instructions are not emulated (FAR CALL/JMP/RET, RET NEAR imm16)
 * - LOOP* and JCXZ opcodes haven't been tested
 * - _jcc_rel32 has been tested only indirectly by _jcc_rel8
 *****************************************************************************/

/*
  Offsets into xde_instr struct.
*/
#define OPCODE  23
#define OPCODE2 24
#define MODRM   25

/*
 Set up our stack and save traced context. The context is saved at the end
 of our stack.
*/
#define SAVE_TRACED_CONTEXT \
    movl %esp, traced_esp   ;\
    movl $stk_end, %esp     ;\
```

```
    pusha                   ;\
    pushf

/*
 Restore traced context from the current top of stack. After that restores
 traced stack pointer.
*/
#define RESTORE_TRACED_CONTEXT \
    popf                    ;\
    popa                    ;\
    movl traced_esp, %esp

/*
 Identity decryption routine. This just copies 8 bytes (BLOCKSIZE) from
 source to destination. Has normal C calling convention. Is not global.
*/
identity_decrypt:
    movl 8(%esp), %edi              /* destination address */
    movl 12(%esp), %esi             /* source address */
    movl $8, %ecx                   /* 8 bytes */
    cld
    rep movsb
    ret

crypt_exec:
.globl crypt_exec
.extern disasm

    /*
     Fetch all arguments. We are called from C and not expected to save
     registers. This is the stack on entry:
     [ ret_addr dfn key lo_addr hi_addr addr ...args ]
    */
    popl %eax                       /* return address */
    popl r_decrypt                  /* real decryption function pointer */
    popl key                        /* encryption key */
    popl lo_addr                    /* low traced eip */
    popl hi_addr                    /* high traced eip */
    popl traced_eip                 /* eip to start tracing */
    pushl %eax                      /* put return address to stack again */

    /*
     now the stack frame resembles as if inner function (starting at
     traced_eip) were called by normal C calling convention (after return
     address, the vararg arguments folow)
    */
    movl %esp, end_esp              /* this is used to stop tracing. */
    movl $0, traced_ctr             /* reset counter of insns to 0 */

decryptloop:
    /*
     This loop traces a single instruction.

     The CONTEXT at the start of each iteration:
     traced_eip: points to the next instruction to execute in traced program

     First what we ever do is switch to our own stack and store the traced
     program's registers including eflags.

     Instructions are encrypted in ECB mode in blocks of 8 bytes. Therefore,
     we always must start decryption at the lower 8-byte boundary. The total
```

```
     of three blocks (24) bytes are decrypted for one instruction. This is
     due to alignment and maximum instruction length constraints: if the
     instruction begins at addres that is = 7 mod 8 + 16 bytes maximum
     length (given some slack) gives instruction span of three blocks.

     Yeah, I know ECB sucks, but this is currently just a proof-of concept.
     Design something better for yourself if you need it.
    */
    SAVE_TRACED_CONTEXT

decryptloop_nocontext:
    /*
     This loop entry point does not save traced context. It is used from
     control transfer instruction emulation where we do all work ourselves
     and don't use traced context.

     The CONTEXT upon entry is the same as for decryptloop.

     First decide whether to decrypt or just trace the plaintext code.
    */
    movl traced_eip, %eax
    movl $identity_decrypt, %ebx    /* assume no decryption */
    cmpl lo_addr, %eax
    jb .store_decrypt_ptr           /* traced_eip < lo_addr */
    cmpl hi_addr, %eax
    ja .store_decrypt_ptr           /* traced_eip > hi_addr */
    movl r_decrypt, %ebx            /* traced_eip in bounds, do decryption */
.store_decrypt_ptr:
    movl %ebx, decrypt

    /*
     Decrypt three blocks starting at eax, reusing arguments on the stack for
     the total of 3 calls. WARNING! For this to work properly, the decryption
     function MUST NOT modify its arguments!
    */
    andl $-8, %eax                  /* round down traced_eip to 8 bytes */
    pushl %eax                      /* src buffer */
    pushl $insn                     /* dst buffer */
    pushl key                       /* key data pointer */
    call *decrypt                   /* 1st block */
    addl $8, 4(%esp)                /* advance dst */
    addl $8, 8(%esp)                /* advance src */
    call *decrypt                   /* 2nd block */
    addl $8, 4(%esp)                /* advance dst */
    addl $8, 8(%esp)                /* advance src */
    call *decrypt                   /* 3rd block */
    addl $12, %esp                  /* clear args from stack */

    /*
     Obtain the real start of instruction in the decrypted buffer. The
     traced eip is taken modulo blocksize (8) and added to the start address
     of decrypted buffer. Then XDE is called (standard C calling convention)
     to get necessary information about the instruction.
    */
    movl traced_eip, %eax
    andl $7, %eax                   /* traced_eip mod 8 */
    addl $insn, %eax                /* offset within decrypted buffer */
    pushl $disbuf                   /* address to disassemble into */
    pushl %eax                      /* instruction offset to disassemble */
    call xde_disasm                 /* fills struct and returns instr length */
    movl %eax, ilen                 /* store instruction length */
```

```
    popl %eax                       /* decrypted insn start */
    popl %ebx                       /* clear remaining arg from stack */

    /*
     Calculate the offset in control table of the instruction handling
     routine. Non-control transfer instructions are just executed in traced
     context, other instructions are emulated.

     Before executing the instruction, the traced eip is advanced by
     instruction length, and the number of executed instructions is
     incremented. We also append indirect 'jmp *continue' after the
     instruction, to continue execution at appropriate place in our tracing.
     The JMP indirect opcodes are 0xFF 0x25.
    */
    movl ilen, %ebx
    addl %ebx, traced_eip           /* advance traced eip */
    incl traced_ctr                 /* increment counter */
    movw $0x25FF, (%eax, %ebx)      /* JMP indirect; little-endian encoding! */
    movl $continue, 2(%eax, %ebx)   /* store address */
    movzbl OPCODE+disbuf, %esi      /* load instruction byte */
    jmp *control_table(,%esi,4)     /* execute by appropirate handler */

.data
    /*
     Emulation routines start here. They are in data segment because code
     segment isn't writable and we are modifying our own code. We don't want
     yet to mess around with mprotect(). One day (non-exec page table support
     on x86-64) it will have to be done anyway..

     The CONTEXT upon entry on each emulation routine:
     eax        : start of decrypted (CURRENT) instruction address to execute
     ilen       : instruction length in bytes
     stack top -> [traced: eflags edi esi ebp esp ebx edx ecx eax]
     traced_esp : original program's esp
     traced_eip : eip of next instruction to execute (NOT of CURRENT insn!)
    */

_unhandled:
    /*
     Unhandled opcodes not normally generated by compiler. Once proper
     emulation routine is written, they become handled :)

     Executing privileged instruction, such as HLT, is the easiest way to
     terminate the program. %eax holds the address of the instruction we were
     trying to trace so it can be observed from debugger.
    */
     hlt

_nonjump:
    /*
     Common emulation for all non-control transfer instructions. Instruction
     buffer (insn) is already filled with decrypted blocks.

     Decrypted instruction can begin in the middle of insn buffer, so the
     relative jmp instruction is adjusted to jump to the traced intruction,
     skipping 'junk' at the beginning of insn.

     When the instruction is executed, our execution continues at location
     where 'continue' points to. Normally, this is decryptloop, but
     occasionaly it is temporarily changed (e.g. in _grp5).
    */
```

```
    subl $insn, %eax                /* instruction begin within insn buffer */
    movb %al, .execute+1            /* update jmp instruction */
    RESTORE_TRACED_CONTEXT
.execute:
    jmp insn                        /* relative, only offset adjusted */
insn:
    .fill 32, 1, 0x90

_jcc_rel8:
    /*
     Relative 8-bit displacement conditional jump. It is handled by relative
     32-bit displacement jump, once offset is adjusted. Opcode must also
     be adjusted: short jumps are 0x70-0x7F, long jumps are 0x0F 0x80-0x8F.
     (conditions correspond directly). Converting short to long jump needs
     adding 0x10 to 2nd opcode.
    */
    movsbl 1(%eax), %ebx            /* load sign-extended offset */
    movb (%eax), %cl                /* load instruction */
    addb $0x10, %cl                 /* adjust opcode to long form */
    /* drop processing to _jcc_rel32 as 32-bit displacement */

_jcc_rel32:
    /*
     Emulate 32-bit conditional relative jump. We pop the traced flags, let
     the Jcc instruction execute natively, and then adjust traced eip
     ourselves, depending whether Jcc was taken or not.

     CONTEXT:
     ebx: jump offset, sign-extended to 32 bits
     cl : real 2nd opcode of the instruction (1st is 0x0F escape)
    */
    movb %cl, ._jcc_rel32_insn+1    /* store opcode to instruction */
    popf                            /* restore traced flags */

._jcc_rel32_insn:
    /*
     Explicit coding of 32-bit relative conditional jump. It is executed
     with the traced flags. Also the jump offset (32 bit) is supplied.
    */
    .byte 0x0F, 0x80
    .long ._jcc_rel32_true - ._jcc_rel32_false

._jcc_rel32_false:
    /*
     The Jcc condition was false. Just save traced flags and continue to
     next instruction.
    */
    pushf
    jmp decryptloop_nocontext

._jcc_rel32_true:
    /*
     The Jcc condition was true. Traced flags are saved, and then the
     execution falls through to the common eip offset-adjusting routine.
    */
    pushf

rel_offset_fixup:
    /*
     Common entry point to fix up traced eip for relative control-flow
     instructions.
```

```
     CONTEXT:
     traced_eip: already advanced to the would-be next instruction. this is
                 done in decrypt_loop before transferring control to any
                 insn-handler.
     ebx       : sign-extended 32-bit offset to add to eip
    */
    addl %ebx, traced_eip
    jmp decryptloop_nocontext

_retn:
    /*
     Near return (without imm16). This is the place where the end-of trace
     condition is checked. If, at this point, esp equals end_esp, this means
     that the crypt_exec would return to its caller.
    */
    movl traced_esp, %ebp           /* compare current traced esp to the */
    cmpl %ebp, end_esp              /* esp when crypt_exec caller's return */
    je ._endtrace                   /* address was on top of the stack */

    /*
     Not equal, emulate ret.
    */
    movl %esp, %ebp                 /* save our current stack */
    movl traced_esp, %esp           /* get traced stack */
    popl traced_eip                 /* pop return address */
    movl %esp, traced_esp           /* write back traced stack */
    movl %ebp, %esp                 /* restore our current stack */
    jmp decryptloop_nocontext

._endtrace:
    /*
     Here the traced context is completely restored and RET is executed
     natively. Our tracing routine is no longer in control after RET.
     Regarding C calling convention, the caller of crypt_exec will get the
     return value of traced function.

     One detail we must watch for: the stack now looks like this:

     stack top -> [ ret_addr ...args ]

     but we have been called like this:

     stack top -> [ ret_addr dfn key lo_addr hi_addr addr ...args ]

     and this is what compiler expects when popping arg list. So we must fix
     the stack. The stack pointer can be just adjusted by -20 instead of
     reconstructing the previous state because C functions are free to
     modify their arguments.

     CONTEXT:
     ebp: current traced esp
    */
    movl (%ebp), %ebx               /* return address */
    subl $20, %ebp                  /* fake 5 extra args */
    movl %ebx, (%ebp)               /* put ret addr on top of stack */
    movl %ebp, traced_esp           /* store adjusted stack */
    RESTORE_TRACED_CONTEXT
    ret                             /* return without regaining control */

    /*
     LOOPNE, LOOPE and LOOP instructions are executed from the common
```

```
     handler (_doloop). Only the instruction opcode is written from
     separate handlers.

     28 is the offset of traced ecx register that is saved on our stack.
    */
_loopne:
    movb $0xE0, ._loop_insn         /* loopne opcode */
    jmp ._doloop
_loope:
    movb $0xE1, ._loop_insn         /* loope opcode */
    jmp ._doloop
_loop:
    movb $0xE2, ._loop_insn         /* loop opcode */
._doloop:
    /*
     * Get traced context that is relevant for LOOP* execution: signed offset,
     * traced ecx and traced flags.
     */
    movsbl 1(%eax), %ebx
    movl 28(%esp), %ecx
    popf

._loop_insn:
    /*
     Explicit coding of loop instruction and offset.
    */
    .byte 0xE0                      /* LOOP* opcodes: E0, E1, E2 */
    .byte ._loop_insn_true - ._loop_insn_false

._loop_insn_false:
    /*
     LOOP* condition false. Save only modified context (flags and ecx)
     and continue tracing.
    */
    pushf
    movl %ecx, 28(%esp)
    jmp decryptloop_nocontext

._loop_insn_true:
    /*
     LOOP* condition true. Save only modified context, and jump to the
     rel_offset_fixup to fix up traced eip.
    */
    pushf
    movl %ecx, 28(%esp)
    jmp rel_offset_fixup

_jcxz:
    /*
     JCXZ. This is easier to simulate than to natively execute.
    */
    movsbl 1(%eax), %ebx            /* get signed offset */
    cmpl $0, 28(%esp)               /* test traced ecx for 0 */
    jz rel_offset_fixup             /* if so, fix up traced EIP */
    jmp decryptloop_nocontext

_callrel:
    /*
     Relative CALL.
    */
```

```
    movb $1, %cl                    /* set to 1 to indicate relative call */
    movl 1(%eax), %ebx              /* get offset */

_call:
    /*
     CALL emulation.

     CONTEXT:
     cl : relative/absolute indicator.
     ebx: absolute address (cl==0) or relative offset (cl!=0).
    */
    movl %esp, %ebp                 /* save our stack */
    movl traced_esp, %esp           /* push traced eip onto */
    pushl traced_eip                /* traced stack */
    movl %esp, traced_esp           /* write back traced stack */
    movl %ebp, %esp                 /* restore our stack */
    testb %cl, %cl                  /* if not zero, then it is a */
    jnz rel_offset_fixup            /* relative call */
    movl %ebx, traced_eip           /* store dst eip */
    jmp decryptloop_nocontext       /* continue execution */

_jmp_rel8:
    /*
     Relative 8-bit displacement JMP.
    */
    movsbl 1(%eax), %ebx            /* get signed offset */
    jmp rel_offset_fixup

_jmp_rel32:
    /*
     Relative 32-bit displacement JMP.
    */
    movl 1(%eax), %ebx              /* get offset */
    jmp rel_offset_fixup

_grp5:
    /*
     This is the case for 0xFF opcode which escapes to GRP5: the real
     instruction opcode is hidden in bits 5, 4, and 3 of the modR/M byte.
    */
    movb MODRM+disbuf, %bl          /* get modRM byte */
    shr $3, %bl                     /* shift bits 3-5 to 0-2 */
    andb $7, %bl                    /* and test only bits 0-2 */
    cmpb $2, %bl                    /* < 2, not control transfer */
    jb _nonjump
    cmpb $5, %bl                    /* > 5, not control transfer */
    ja _nonjump
    cmpb $3, %bl                    /* CALL FAR */
    je _unhandled
    cmpb $5, %bl                    /* JMP FAR */
    je _unhandled
    movb %bl, %dl                   /* for future reference */

    /*
     modR/M equals 2 or 4 (near CALL or JMP).
     In this case the reg field of modR/M (bits 3-5) is the part of
     instruction opcode.

     Replace instruction byte 0xFF with 0x8B (MOV r/m32 to reg32 opcode).
     Replace reg field with 3 (ebx register index).
    */
```

```
    movb $0x8B, (%eax)          /* replace with MOV_to_reg32 opcode */
    movb 1(%eax), %bl           /* get modR/M byte */
    andb $0xC7, %bl             /* mask bits 3-5 */
    orb $0x18, %bl              /* set them to 011=3: ebx reg index */
    movb %bl, 1(%eax)           /* set MOV target to ebx */

    /*
     We temporarily update continue location to continue execution in
     this code instead of jumping to decryptloop. We execute MOV in TRACED
     context because it must use traced registers for address calculation.
     Before that we save OUR esp so that original TRACED context isn't lost
     (MOV updates ebx, traced CALL wouldn't mess with any registers).

     First we save OUR context, but after that we must restore TRACED ctx.
     In order to do that, we must adjust esp to point to traced context
     before restoration.
    */
    movl $._grp5_continue, continue
    movl %esp, %ebp             /* save traced context pointer into ebp */
    pusha                       /* store our context; eflags irrelevant */
    movl %esp, our_esp          /* our context pointer */
    movl %ebp, %esp             /* adjust traced context pointer */
    jmp _nonjump

._grp5_continue:
    /*
     This is where execution continues after MOV calculates effective
     address for us.

     CONTEXT upon entry:
     ebx: target address where traced execution should continue
     dl : opcode part (bits 3-5) of modR/M, shifted to bits 0-2
    */
    movl $decryptloop, continue     /* restore continue location */
    movl our_esp, %esp              /* restore our esp */
    movl %ebx, 16(%esp)             /* so that ebx is restored anew */
    popa                            /* our context along with new ebx */
    cmpb $2, %dl                    /* CALL near indirect */
    je ._grp5_call
    movl %ebx, traced_eip           /* JMP near indirect */
    jmp decryptloop_nocontext
._grp5_call:
    xorb %cl, %cl                   /* mark that addr in ebx is absolute */
    jmp _call

_0xf:
    /*
     0x0F opcode esacpe for two-byte opcodes. Only 0F 0x80-0x8F range are
     Jcc rel32 instructions. Others are normal instructions.
    */
    movb OPCODE2+disbuf, %cl        /* extended opcode */
    cmpb $0x80, %cl
    jb _nonjump                     /* < 0x80, not Jcc */
    cmpb $0x8F, %cl
    ja _nonjump                     /* > 0x8F, not Jcc */
    movl 2(%eax), %ebx              /* load 32-bit offset */
    jmp _jcc_rel32

control_table:
    /*
     This is the jump table for instruction execution dispatch. When the real
```

```
 opcode of the instruction is found, the tracer jumps indirectly to
 execution routine based on this table.
*/
.rept 0x0F                      /* 0x00 - 0x0E */
.long _nonjump                  /* normal opcodes */
.endr
.long _0xf                      /* 0x0F two-byte escape */

.rept 0x60                      /* 0x10 - 0x6F */
.long _nonjump                  /* normal opcodes */
.endr

.rept 0x10                      /* 0x70 - 0x7F */
.long _jcc_rel8                 /* relative 8-bit displacement */
.endr

.rept 0x10                      /* 0x80 - 0x8F */
.long _nonjump                  /* long displacement jump handled from */
.endr                           /* _0xf opcode escape */

.rept 0x0A                      /* 0x90 - 0x99 */
.long _nonjump
.endr
.long _unhandled                /* 0x9A: far call to full pointer */
.rept 0x05                      /* 0x9B - 0x9F */
.long _nonjump
.endr

.rept 0x20                      /* 0xA0 - 0xBF */
.long _nonjump
.endr

.long _nonjump, _nonjump        /* 0xC0, 0xC1 */
.long _unhandled                /* 0xC2: retn imm16 */
.long _retn                     /* 0xC3: retn */
.rept 0x06                      /* 0xC4 - 0xC9 */
.long _nonjump
.endr
.long _unhandled, _unhandled    /* 0xCA, 0xCB : far ret */
.rept 0x04
.long _nonjump
.endr

.rept 0x10                      /* 0xD0 - 0xDF */
.long _nonjump
.endr

.long _loopne, _loope           /* 0xE0, 0xE1 */
.long _loop, _jcxz              /* 0xE2, 0xE3 */
.rept 0x04                      /* 0xE4 - 0xE7 */
.long _nonjump
.endr
.long _callrel                  /* 0xE8 */
.long _jmp_rel32                /* 0xE9 */
.long _unhandled                /* far jump to full pointer */
.long _jmp_rel8                 /* 0xEB */
.rept 0x04                      /* 0xEC - 0xEF */
.long _nonjump
.endr
```

```
    .rept 0x0F                      /* 0xF0 - 0xFE */
    .long _nonjump
    .endr
    .long _grp5                     /* 0xFF: group 5 instructions */

.data
continue:   .long decryptloop       /* where to continue after 1 insn */

.bss
.align  4
traced_esp: .long 0                 /* traced esp */
traced_eip: .long 0                 /* traced eip */
traced_ctr: .long 0                 /* incremented by 1 for each insn */
lo_addr:    .long 0                 /* low encrypted eip */
hi_addr:    .long 0                 /* high encrypted eip */
our_esp:    .long 0                 /* our esp... */
end_esp:    .long 0                 /* esp when we should stop tracing */
local_stk:  .fill 1024, 4, 0        /* local stack space (for calling C) */
stk_end = .                         /* we need this.. */
ilen:       .long 0                 /* instruction length */
key:        .long 0                 /* pointer to key data */
decrypt:    .long 0                 /* actually USED decryption function */
r_decrypt:  .long 0                 /* REAL decryption function */
disbuf:     .fill 128, 1, 0         /* xde disassembly buffer */
```

# 9 The file encryption utility source: cryptexec.c

```
/*
Copyright (c) 2004 Zeljko Vrba

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to permit persons to
whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

/*
 * This program encrypts a portion of the file, writing new file with
 * .crypt appended. The permissions (execute, et al) are NOT preserved!
 * The blocksize of 8 bytes is hardcoded.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "cast5.h"
```

```
#define BLOCKSIZE 8
#define KEYSIZE 16

typedef void (*cryptblock_f)(void*, u8*, const u8*);

static unsigned char *decode_hex_key(char *hex)
{
    static unsigned char key[KEYSIZE];
    int i;

    if(strlen(hex) != KEYSIZE << 1) {
        fprintf(stderr, "KEY must have EXACTLY %d hex digits.\n", KEYSIZE << 1);
        exit(1);
    }

    for(i = 0; i < KEYSIZE; i++, hex += 2) {
        unsigned int x;
        char old = hex[2];

        hex[2] = 0;
        if(sscanf(hex, "%02x", &x) != 1) {
            fprintf(stderr, "non-hex digit in KEY.\n");
            exit(1);
        }
        hex[2] = old;
        key[i] = x;
    }

    return key;
}

static void *docrypt(
    FILE *in, FILE *out,
    long startoff, long endoff,
    cryptblock_f crypt, void *ctx)
{
    char buf[BLOCKSIZE], enc[BLOCKSIZE];
    long curroff = 0;
    size_t nread = 0;

    while((nread = fread(buf, 1, BLOCKSIZE, in)) > 0) {
        long diff = startoff - curroff;

        if((diff < BLOCKSIZE) && (diff > 0)) {
            /*
            this handles the following mis-alignment (each . is 1 byte)
            ...[..|......]....
                ^  ^      ^ curoff+BLOCKSIZE
                |  startoff
             curroff
            */
            if(fwrite(buf, 1, diff, out) < diff) {
                perror("fwrite");
                exit(1);
            }
            memmove(buf, buf + diff, BLOCKSIZE - diff);
            fread(buf + BLOCKSIZE - diff, 1, diff, in);
            curroff = startoff;
        }
```

```
        if((curroff >= startoff) && (curroff < endoff)) {
            crypt(ctx, enc, buf);
        } else {
            memcpy(enc, buf, BLOCKSIZE);
        }
        if(fwrite(enc, 1, nread, out) < nread) {
            perror("fwrite");
            exit(1);
        }
        curroff += nread;
    }
}

int main(int argc, char **argv)
{
    FILE *in, *out;
    long startoff, endoff;
    char outfname[256];
    unsigned char *key;
    struct cast5_ctx ctx;
    cryptblock_f mode;

    if(argc != 6) {
        fprintf(stderr, "USAGE: %s <-e|-d> FILE KEY STARTOFF ENDOFF\n", argv[0]);
        fprintf(stderr, "KEY MUST be 32 hex digits (128 bits).\n");
        return 1;
    }

    if(!strcmp(argv[1], "-e")) {
        mode = cast5_encrypt;
    } else if(!strcmp(argv[1], "-d")) {
        mode = cast5_decrypt;
    } else {
        fprintf(stderr, "invalid mode (must be either -e od -d)\n");
        return 1;
    }

    startoff = atol(argv[4]);
    endoff = atol(argv[5]);
    key = decode_hex_key(argv[3]);

    if(cast5_setkey(&ctx, key, KEYSIZE) < 0) {
        fprintf(stderr, "error setting key (maybe invalid length)\n");
        return 1;
    }

    if((endoff - startoff) & (BLOCKSIZE-1)) {
        fprintf(stderr, "STARTOFF and ENDOFF must span an exact multiple of %d bytes\n", BLOCKSIZE);
        return 1;
    }
    if((endoff - startoff) < BLOCKSIZE) {
        fprintf(stderr, "STARTOFF and ENDOFF must span at least %d bytes\n", BLOCKSIZE);
        return 1;
    }

    sprintf(outfname, "%s.crypt", argv[2]);
    if(!(in = fopen(argv[2], "r"))) {
        fprintf(stderr, "fopen(%s): %s\n", argv[2], strerror(errno));
        return 1;
    }
    if(!(out = fopen(outfname, "w"))) {
```

```
        fprintf(stderr, "fopen(%s): %s\n", outfname, strerror(errno));
        return 1;
    }

    docrypt(in, out, startoff, endoff, mode, &ctx);

    fclose(in);
    fclose(out);
    return 0;
}
```

# 10 The test program: test2.c

```
/*
Copyright (c) 2004 Zeljko Vrba

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to permit persons to
whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "cast5.h"

#define BLOCKSIZE 8
#define KEYSIZE 16

/*
 * f1 and f2 are encrypted with the following 128-bit key:
 * 5f4dcc3b5aa765d61d8327deb882cf99 (MD5 of the string 'password')
 */

static int f1(int a)
{
    int i, s = 0;

    for(i = 0; i < a; i++) {
        s += i*i;
    }
    printf("called plaintext code: f1 = %d\n", a);
    return s;
}
```

```
static int f2(int a, int b)
{
    int i;

    a = f1(a);
    for(i = 0; i < b; i++) {
        a += b;
    }
    return a;
}

static unsigned char *decode_hex_key(char *hex)
{
    static unsigned char key[KEYSIZE];
    int i;

    if(strlen(hex) != KEYSIZE << 1) {
        fprintf(stderr, "KEY must have EXACTLY %d hex digits.\n", KEYSIZE << 1);
        exit(1);
    }

    for(i = 0; i < KEYSIZE; i++, hex += 2) {
        unsigned int x;
        char old = hex[2];

        hex[2] = 0;
        if(sscanf(hex, "%02x", &x) != 1) {
            fprintf(stderr, "non-hex digit in KEY.\n");
            exit(1);
        }
        hex[2] = old;
        key[i] = x;
    }

    return key;
}

int main(int argc, char **argv)
{
    int a, b, result;
    char op[16], hex[256];
    void *esp;
    struct cast5_ctx ctx;

    printf("enter decryption key: ");
    scanf("%255s", hex);
    if(cast5_setkey(&ctx, decode_hex_key(hex), KEYSIZE) < 0) {
        fprintf(stderr, "error setting key.\n");
        return 1;
    }

    printf("a b = "); scanf("%d %d", &a, &b);

    asm("movl %%esp, %0" : "=m" (esp));
    printf("esp=%p\n", esp);
    result = crypt_exec(cast5_decrypt, &ctx, f1, decode_hex_key, f2, a, b);
    asm("movl %%esp, %0" : "=m" (esp));
    printf("esp=%p\n", esp);
    printf("result = %d\n", result);
```

```
    return 0;
}
```