9 Davinci Seal: Self-decrypting Executables

by Ryan O'Neill,
who also publishes as Elfmaster

In the pursuit of creativity and fun, I recently had the idea of creating self-protecting files. That is to say, any type of data that you want protected from analysis, with the ability to decrypt its own content when provided the correct key. The use cases for such a capability are debatable, but the idea is nevertheless fun, and only took an afternoon to implement. The goal was to create a program that can transform any file into an ELF executable whose sole purpose is protecting the file data embedded within its own body. I call these Davinci Seals.

9.1 Protection

The output executable should be able to protect the embedded data from static analysis and resist runtime analysis and `ptrace`-based debugging. An attacker should not be able to extract the content by setting breakpoints and reading the decrypted content from memory; thus, detection of such attacks should be in place. The executable should also be resistant to attackers modifying code or replacing anti-debug code with NOP instructions; this can be mostly prevented by using code watermarking. There are forms of dynamic analysis such as dynamic instrumentation with Pin, or using an IDA Emulator plugin, which Davinci does not mitigate, but we briefly discuss viable methods for protection against them.

9.2 Example of creating a Davinci seal

```
$ cat msg.txt
______________________
|The spice must flow |
----------------------

$ ./davinci msg.txt msg.dvs p4ssw0rd -r
[+] The user who executes msg.dvs must supply password: p4ssw0rd
[+] Encoding payload data
[+] Encoding payload struct
[+] Building msg program
[+] (Optional) utils/stripx exists, so using it to strip section headers off of DRM archive
Successfully created msg.dvs

** NOTE: msg.txt was transformed into an ELF executable (A davinci seal) named msg.dvs

$ readelf -l msg.dvs

Elf file type is EXEC (Executable file)
Entry point 0x400492
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000918 0x0000000000000918  R E    200000
  LOAD           0x0000000000001000 0x0000000000601000 0x0000000000601000
                 0x0000000000800324 0x0000000000800338  RW     200000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000024 0x0000000000000024  R      4
  GNU_EH_FRAME   0x00000000000006c0 0x00000000004006c0 0x00000000004006c0
                 0x000000000000007c 0x000000000000007c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

```

```
$ ./msg.dvs
This message requires that you supply a key to decrypt

$ ./msg.dvs p4ssw0rd
______________________
|The spice must flow |
______________________
```

Voila! Our msg.txt file was transformed into msg.dvs, an ELF executable which lives and breathes only to protect the data within it, and reveal that data when supplied the encryption key.

9.3 Implementation

9.3.1 ELF stub and payload packaging

The goal here is to transform a file containing arbitrary data into an ELF executable whose sole purpose is to protect the data. The executable should decrypt and write the data to stdout if the proper password/key is supplied.

Our project consists of two parts. The first is the Protector, which creates the output program from the second, which we'll call the Stub.

The protector program takes an input file and generates a stub executable that contains the encrypted input file within it, as well as metadata describing the size and location of the data. The stub executable that it generates is written mostly in C, then compiled into bytecode and stored within the protector executable. To fully understand the protector, we must first understand the stub.

The basic principle of the stub is that it contains an encrypted file. This encrypted data must be stored somewhere with information about it. The best way to implement this is to append the data to the data segment of the stub executable, or even within the text segment using a reverse extension method. Both methods are common in virus infection and executable packers, but for the sake of POC and simplicity we will pre-allocate a fixed size within the initialized data section of the stub executable.

```c
/* From davinci.h */
#define KEY_BUF_LEN 256
#define MAX_PAYLOAD_SIZE ((1024 * 1024) * 8)

typedef struct payload_meta {
        uint64_t payload_len;      /* Length of the encrypted file data */
        uint32_t keylen;           /* Length of the key used to encrypt */
        uint8_t key[KEY_BUF_LEN];    /* The key used to encrypt/decrypt */
        uint8_t data[MAX_PAYLOAD_SIZE];   /* The file data itself */
} payload_meta_t;

/* From stub.c */
payload_meta_t payload __attribute__((section(".data"))) = {0x0};
```

Since the data and metadata will be stored in the structure above, the protector can resolve the `payload` symbol to find where it needs to store the file data and key data within the stub.

```
-- Illustration of the work flow:

[input file (msg.txt)] /* The input file can be anything */
     |
     v
[protector] /* This program transforms msg.txt into msg.elf */
     |
     v
[output file(msg.elf)] /* The output is a compiled stub.c, instrumented with the encrypted
     input file , and metadata */
```

9.3.2 Anti-analysis protection

The goal is to transform an input file into an output executable that protects it. The input file is encrypted/obfuscated and embedded within an ELF executable that serves as a defensive shell. This defensive shell will decrypt the data if supplied the correct key, and write it to standard output. If you choose, you may tell the protector to store an obfuscated copy of the key within the binary so that it decrypts itself without a supplied password. This offers no real protection, of course, but may still have some application.

Our defensive shell, being an executable and all, is inherently vulnerable to reverse engineering, static analysis, and debugging (dynamic analysis) attacks. It would behoove the defending binary to have some protection against some of these attacks. We have three protections against static analysis:

1.) The body of the input file is encrypted within the output executable, though just with weak XOR for this proof of concept. The `payload_meta_t` structure is also encrypted, on top of the `payload.data` buffer. If Davinci is to become more than just a proof of concept, a real cipher must be used.

2.) The section header table is stripped from the ELF executable. String tables are zeroed out, and the symbol table is discarded.

This by itself makes the output executable far more difficult to navigate with a disassembler, since there is no information provided about symbols or specific sections. The program headers are suitable for loading and running a program, but without section headers, the program is more difficult to analyze, even for IDA Pro.

Stripping the ELF section headers effectively disables any tools that rely on section headers. It is an old and simple technique used by many neighbors.

```
--Prevents objdump disassembly
$ objdump -D msg.dvs
msg.dvs:      file format elf64-x86-64
$

--Prevents symbol lookups
$ readelf -s msg.dvs
$
```

3.) The output executable is further protected with UPX, the Ultimate Packer for eXecutables. This also takes care of shrinking the executable from the wasteful fixed-size of our buffer.

This feature is primarily for shrinking the output executable, because the stub is by default fixed at a large size. Initializing an 8 MB buffer in the `.data` section leaves room for files up to 8 MB. As mentioned earlier, another method, such as appending to the data segment, would be a better long-term design decision and would result in the executable growing in proportion to the input file size. For the sake of POC, we used the method of initializing fixed space in the `.data` section, which allows us to focus more on the principles and less on the implementation.

9.3.3 Anti-debugging tricks

Most debuggers, such as GDB, rely on the `ptrace` system call. If `ptrace`-based debugging can be prevented, we eliminate the most common types of dynamic analysis tools. `strace`, `gdb`, dumping `/proc/$pid/mem`, and other tricks will all break.

Anti-Ptrace Protection A process is only allowed to have one tracer. This means that we can simply use `ptrace` within our stub executable, so that it traces itself, preventing any other debuggers/tracers from attaching. If a debugger is attached before our stub calls `ptrace()`, then our call to `ptrace()` will return `-1` and we can abort the process.

The `enable_anti_debug()` function will prevent `gdb` and `strace` from analyzing our ELF executable.

```
/*
 * Notice that we use our own wrapper for the ptrace syscall.
 * This is good practice to prevent LD_PRELOAD bypasses --
 * even though our stub is compiled -nostdlib (in which case
 * an LD_PRELOAD bypass would not work anyway).
 */

static long _ptrace(long request, long pid, void *addr, void *data) {
        long ret;

        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov %1, %%rsi\n"
                        "mov %2, %%rdx\n"
                        "mov %3, %%r10\n"
                        "mov $101, %%rax\n"
                        "syscall" : : "g"(request), "g"(pid), "g"(addr), "g"(data));
        asm("mov %%rax, %0" : "=r"(ret));

        return ret;
}

void bail_out(void) {
        _write(1, "The gates of heaven remain closed\n", 34);
        _kill(_getpid(), SIGKILL);
        __exit(-1);
}

void enable_anti_debug(void) {
        if (_ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
                bail_out(); // if a debugger is already attached we bail out
        // a marker showing that an attacker didn't just jump over enable_anti_debug()
        data_watermark++;
}
```

Now what happens when we try to debug `msg.dvs` with `gdb`?

```
$ gdb -q msg.dvs
Reading symbols from msg.dvs...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ryan/dev/davinci/msg.dvs
The gates of heaven remain closed
Program terminated with signal SIGKILL, Killed.
The program no longer exists.
(gdb)
```

If an attacker wants to bypass the anti-`ptrace` code, there are several techniques that are commonly used.

1. `LD_PRELOAD` can be used to preload a library. This loads the specified library before any others, and any of its symbols will take precedence over subsequently loaded libraries. Attackers have used this to preload a custom shared library with a dummy `ptrace` that simply returns success and does nothing. In our stub executable we do not use dynamic linking, and therefore no shared libraries can even be loaded. We also use a syscall wrapper for `ptrace`, so that even if our stub did use dynamic linking, our calls to `ptrace` would not go through the PLT/GOT and therefore could not be hijacked with another shared library call. Always use syscall wrappers in binary hardening code, and stay away from glibc.

2. An attacker could modify the stub's binary code so that the `enable_anti_debug()` code is never called, or simply jumped over. An attacker could also overwrite the code in `enable_anti_debug()` so that it doesn't actually do anything to prevent debugging. We use a simple form of code watermarking to try to prevent this, which we will discuss in Section 9.3.4.

/proc/<pid>/mem Dump Protection It is a common practice for reverse engineers/attackers to dump a hardened binary from memory. This can be done by attaching to the process and reading `/proc/<pid>/mem`. If the process is already stopped, then attaching to the process isn't necessary, and a simple `read()` suffices. Fortunately, Linux has a neat syscall called `prctl()`, which allows us to change the characteristics of our running programs, but must be issued by the program itself.

```
      int prctl(int option, unsigned long arg2, unsigned long arg3,
                unsigned long arg4, unsigned long arg5);

  OPTION: PR_SET_DUMPABLE (since Linux 2.3.20)
          Setting arg2 to 0
     prevents process from dumping a CORE file,
     prevents process from being attached to with ptrace, and
     prevents process from being dumped from /proc/<pid>/mem.
```

The `PR_SET_DUMPABLE` option applies several very neat and useful anti-debugging features. We use this to add even more resistance to `ptrace`, while also preventing core dumps and memory dumps of our process.

```
/*
 * Always implement a syscall wrapper when using syscalls for anti-debugging
 */
int _prctl(long option, unsigned long arg2, unsigned long arg3,
           unsigned long arg4, unsigned long arg5) {
        long ret;

        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov %1, %%rsi\n"
                        "mov %2, %%rdx\n"
                        "mov %3, %%r10\n"
                        "mov $157, %%rax\n"
                        "syscall\n" :: "g"(option), "g"(arg2), "g"(arg3),
                                       "g"(arg4), "g"(arg5));
        asm("mov %%rax, %0" : "=r"(ret));
        return (int)ret;
}

/*
 * Simply call _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0) from your code.
 * (Ideally from a glibc constructor)
 */

void anti_debug_dump(void) __attribute__ ((constructor));

void anti_debug_dump(void) {
  _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0);
}
```

SIGTRAP Detection When breaking binaries, the attacker generally will set breakpoints in specific areas of the code. With `SIGTRAP` detection we can detect breakpoints, as they generate a `SIGTRAP` signal. Upon detection we can do whatever we like, ideally bail out and kill the program.

This can be done by creating a signal handler for SIGTRAP. If our signal handler catches the signal, then it means there is no debugger attached. Since our stub is not linked to libc in any way, we must use our own syscall wrapper for sigaction. Thanks to Jpanic for pointing out important caveats that must be considered when doing this.

```c
#define SA_RESTORER 0x04000000

/* struct sigaction act.sa_restorer must point to a handler
 * that performs an rt_sigreturn(0)-- normally this is done
 * by glibc.
 */
int _sigreturn(unsigned long unused) {
        unsigned long ret;
        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov $15, %%rax\n"
                        "syscall" : : "g"(unused));
        __asm__("mov %%rax, %0" : "=r"(ret));
        return (int)ret;
}

/* We increment trap_count if we caught the signal */
int trap_count = 0;

void sigcatch(int sig) {
        trap_count++;
}

/* This function sets up a signal handler for SIGTRAP
 * if a debugger caught it.
 */

void install_trap_handler(void) {
        struct sigaction act, oldact;
        act.sa_handler = sigcatch;
        act.sa_flags = SA_RESTORER;
        act.sa_restorer = restore;
        sigemptyset(&act.sa_mask);
        sigaddset(&act.sa_mask, SIGTRAP);
        // must pass sizeof(long) or kernel returns -EINVAL
        _sigaction (SIGTRAP, &act, NULL, sizeof(long));

}

void detect_debugger(void) {
  __asm__ ("int3\n"
     "nop");
  if (trap_count == 0)
    bail_out(); // debugger caught the trap, bail out!
  trap_count = 0;
}
```

There exist other anti-debugging techniques not used in this example. /proc/self/status can check if a ptrace attachment exists. Junk or misaligned assembly code could be used to obfuscate the application against a disassembler while keeping it functionally equivalent.

Advanced reverse engineers will go well beyond the use of ptrace()-based debuggers when attempting dynamic analysis. Such engineers might use the Pin instrumentation framework, an emulator, or ERESI's e2dbg.

Detection of Pin hooking can be done by checking /proc/self/maps to see whether the mapping called [vvar] exists after [vdso]. This happens when vdso has been partially remapped by Pin.

Emulation detection can also be performed by rtdsc timestamp checking.

9.3.4 Code and data watermarking

To enforce our anti-debugging code so that it is not easily circumvented, we have some simple code and data watermarking in-place. As mentioned earlier, if someone were to modify the `enable_anti_debug()` code, or simply jump over it, it would be rendered useless. We must therefore be prepared to detect when this happens and act accordingly by exiting or killing the program before it is successfully cracked.

Data Watermarking For the data watermarking, we have a static initialized variable that is set to 0 and only incremented after the `enable_anti_debug()` function successfully completes. Later on, we check the value of this variable. If it has not been incremented, then we can assume that an attacker either jumped over the anti-debug code or NOP'd it out.

```
void denied(void) {
        bail_out();
}

void accepted(void) {
        __asm__ __volatile__("nop\n");
}

_start() {
  uint64_t a[2], x;
        void (*f)();
  int ret;

  ... <code> ...

  a[0] = (uint64_t)&denied;    // a[0] points to denied() address
        a[1] = (uint64_t)&accepted;    // a[1] points to accepted() address
        x = a[!(!(data_watermark))];   // convert data_watermark to a boolean, 0 or 1
        f = (void *)x;           // assign function pointer to either accepted() or denied()
        f();         // call accepted() or denied()

  ... <code> ...
}
```

As we can see by the code snippet above, if `data_watermark` was not incremented it will still be 0, so we can assume that an attacker jumped over the `enable_anti_debug()` code. So `denied()` would be called, which calls `bail_out()` to kill the process. Otherwise, `accepted()` will be called, which does nothing, and our binary goes on running untampered.

Code Watermarking For the code watermarking, we want to validate that the `enable_anti_debug()` function has not been modified in any way. We do this by simply fingerprinting it.

```
/* From davinci.h */
typedef struct code_watermark {
        uint32_t code_size;
        uint8_t code_signature[CODE_CHUNK_SIZE];
} code_watermark_t;


/* From davinci.c
 * NOTE: 'uint8_t *mem is a mapping of the stub executable'
 * This code will create the fingerprint of enable_anti_debug() and store
 * it within the stub executable
 */
 ... <code> ...

  symval = resolve_symbol("enable_anti_debug", mem);
  symsize = resolve_symbol_size("enable_anti_debug", mem);
  offset = textOffset + (symval - textVaddr);
  code_watermark = (code_watermark_t *)alloca(sizeof(code_watermark_t));
  memcpy((uint8_t *)code_watermark->code_signature, (uint8_t *)&mem[offset], symsize);
  code_watermark->code_size = symsize;
  symval = resolve_symbol("code_watermark", mem);
  symsize = resolve_symbol_size("code_watermark", mem);
  offset = dataOffset + (symval - dataVaddr);
  memcpy((void *)&mem[offset], (void *)code_watermark, sizeof(code_watermark_t));
 ... <code> ...

/* From stub.c
 * We memcmp the enable_anti_debug() function with code_watermark.code_signature.
 * If there are any discrepancies, we call denied(), which bails out and prints the message
 * "The gates of heaven remain closed"
 */
 ... <code> ...

         a[0] = (uint64_t)&accepted;
         a[1] = (uint64_t)&denied;
         ret = _memcmp((uint8_t *)code_watermark.code_signature, (uint8_t *)enable_anti_debug
    , code_watermark.code_size);
         x = a[!(!(ret))];
         f = (void *)x;
         f();
 ... <code> ...
```

9.4 Getting Davinci

The Davinci source code tarball is stored in a davinci seal itself :)

```
chmod +x davinci.tgz.dvs
./davinci.tgz.dvs d4v1nc1 > davinci.tgz
tar zxvf davinci.tgz
```

Secrets of ELF Now Revealed!
Easy to learn RESULTS GUARANTEED. Home Study Course. ELF BINARY WORKSHOP
Seattle, WA
2015 January 8th to 9th
http://0x7f.eventbrite.com
"BE THERE"
Key components
ELF reverse engineering
ELF forensic analysis
ELF virus design
ELF binary patching
ELF anti-forensics
ELF core concepts
Brought to you by Elfmaster & Leviathan Security

"For the last time, Brian," said Barbie, "`$4C` is absolute jump and `$6C` is indirect jump. It's like this: `$4C` is me telling you that you're an idiot; `$6C` is me pointing you to a piece of paper that says, 'You're an idiot.' And what the hell are you smiling at, Steven? You've got code here that overwrites the ROM monitor. Unless your last name is Wozniak, STFO out of `$F000` block."