Leviathan Research

Extending the ELF Core Format for Forensics Snapshots

Ryan O'Neill
Security Consultant

10 November 2014

limitless innovation. no compromise

Contents

1 Abstract

Malware authors often apply advanced techniques to execute and hide malicious activity on victim Linux systems. ELF runtime infections are among those techniques; they mutate the memory of a victim process to then modify its workings while maintaining stealth against disk-based forensics.

To detect ELF runtime infections, analysts combine live debugging of the victim process and dumping a core file or taking a memory snapshot for offline analysis. This pervasive approach and the various methods used for detection are unwieldy and were never intended for use in forensic investigation; they demand room for improvement.

In this paper I describe my Extended Core File Snapshot (ECFS) format, an extension to the Linux ELF core file format. I show that ECFS accurately captures all relevant, in-memory forensic information necessary for an analyst to diagnose common ELF memory injection attacks, and I explain its application in real-world scenarios. ECFS allows for both offline debugger (GDB) analysis of snapshot files and interoperability with pervasive static ELF Binary analysis tools. In addition to capturing available information via existing techniques, such as with Volatility [9] process dumps, ECFS uses a novel technique to aid analysis by identifying dynamic symbols and local function entry points that are generally stripped away in production binaries.

I will provide an open source implementation of ECFS and supporting tools in January of 2015.

2 Introduction

The actors behind UNIX-based malware attacks have a diverse set of goals, from bitcoin mining to staging targeted attacks against large institutions. While not all malware requires the use of sophisticated methods, the majority of traditional malware attacks aim to achieve stealth, persistence, and elevation of privilege by exploiting resident components of the victim system.

Perhaps the most well publicized technique for opening a malicious backdoor in a Linux machine is via the notorious kernel rootkit. However, in cases where installing a kernel rootkit isn't feasible or desired, attackers are increasingly looking towards other techniques such as ELF runtime infections.

ELF runtime infections infect process memory in an attempt to backdoor or modify the workings of a running process, often without requiring root privileges while evading disk-based forensics techniques. There are very few types of software that are capable of ascertaining the complexities and nuances of these ELF runtime infections and performing the analysis by hand is a resource-consuming task, even for the expert. As it stands there are approximately four standard approaches used by defenders to analyze the memory of a running process for ELF runtime infections:

1. Use the *ptrace* system call or *ptrace*-assisted debugger such as GDB to analyze a running instance of the suspect process
2. Dump /proc/<pid>/mem into a file for offline analysis
3. Generate a core file for offline analysis with GDB
4. Generate a process memory dump using Volatility for analysis

Ideally, the dynamic and static analysis techniques listed above are used in combination to identify ELF runtime infections, but sometimes the realities of forensic analysis requires using them in isolation. Specifically, it may not be feasible to leave a victim process running or to maintain access to a victim Linux system to allow thorough dynamic analysis. Even when both static analysis and dynamic analysis of a victim process are possible, automating forensics analysis of existing information in its various locations and formats can be cumbersome. My research and this paper provide tools to address these shortcomings. The ECFS format discussed in this paper was originally designed as the native snapshot format for VMA Vudu [1], which is a process memory and binary forensics analysis tool for Linux that also may also act as a host-based intrusion detection system.

While I designed ECFS specifically as the file format for VMA Vudu [1], it became clear during its development that it could also be instrumental for manual offline forensics analysis of process memory and that a broader audience could make use of it with Linux *binutils* [10]. This is because ECFS is compatible with common ELF specifications, and includes detailed information about a respective process. The forensic-relevant information is stored in such a way that common GNU utilities such as *objdump* and *readelf* will suffice for analyzing an ECFS snapshot when looking for the presence of an attacker within a process. ECFS modifies the ELF design of the core file format to support both storing of forensic-relevant information and analysis via ELF binary analysis tools. As a core file, an ECFS file can be readily analyzed via a debugger (such as GDB) after modifying a single field in its ELF header. Otherwise it is readable via any tool capable of viewing ELF section headers, program headers, and symbol tables. The ECFS tool, which I specifically designed to create process memory snapshot files, has been forked from VMA Vudu and is maintained as a standalone project.

2.1 ELF Core Files (ET_CORE)

Currently every UNIX flavor operating system that supports ELF adopts the ELF core file format and will generate a core file upon signaling a process with signal 11 *SIGSEGV*. The output file is a perfect snapshot of all the memory mappings from the process image with an ELF header of type ET_CORE. In the PT_NOTE segment a plethora of information is stored about what shared libraries are mapped, the CPU registers, signal information, process state, and personality. These are all stored as ELF notes. The ELF notes segment is a good place to pass auxiliary information and is the location that GDB looks to find this data when debugging a core file. This data can be viewed in a core file by reading the notes section with the *readelf* utility. Each OS will have a slight variation of the way this information is stored in the notes of the file.

The notes from a regular Core file in Linux:

```
ryan@elfmaster:~$ readelf -n core
Displaying notes found at file offset 0x00000430 with length 0x00000b74:
  Owner                 Data size     Description
  CORE                 0x00000150     NT_PRSTATUS (prstatus structure)
  CORE                 0x00000088     NT_PRPSINFO (prpsinfo structure)
  CORE                 0x00000080     NT_SIGINFO (siginfo_t data)
  CORE                 0x00000130     NT_AUXV (auxiliary vector)
  CORE                 0x0000021f     NT_FILE (mapped files)
    Page size: 4096
    Start                 End                   Page Offset
    0x0000000000400000  0x0000000000401000  0x0000000000000000
          /home/ryan/host
    0x0000000000600000  0x0000000000601000  0x0000000000000000
        /home/ryan/host
    0x0000000000601000  0x0000000000602000  0x0000000000000001
        /home/ryan/host
    0x00007f579d462000  0x00007f579d61d000  0x0000000000000000

/lib/x86_64-linux-gnu/libc-2.19.so
    0x00007f579d61d000  0x00007f579d81d000  0x00000000000001bb

        /lib/x86_64-linux-gnu/libc-2.19.so
    0x00007f579d81d000  0x00007f579d821000  0x00000000000001bb
        /lib/x86_64-linux-gnu/libc-2.19.so
    0x00007f579d821000  0x00007f579d823000  0x00000000000001bf
        /lib/x86_64-linux-gnu/libc-2.19.so
    0x00007f579d828000  0x00007f579d84b000  0x0000000000000000
        /lib/x86_64-linux-gnu/ld-2.19.so
    0x00007f579da4a000  0x00007f579da4b000  0x0000000000000022
        /lib/x86_64-linux-gnu/ld-2.19.so
  CORE                0x00000200     NT_FPREGSET (floating point registers)
  LINUX               0x00000340     NT_X86_XSTATE (x86 XSAVE extended state)
```

From this readelf output you can see that the prstatus structure, the prpsinfo structure, and siginfo_t are stored in the notes. The auxiliary vector contains information about the setup of the stack at program loading time. The Linux source code files fs/binfmt_elf.c and fs/binfmt_fdpic_elf.c contain the code that handle all ELF loading and core dumps.

For every memory mapping in the process image that is dumped into the ELF core file there is a program header in the program header table. This includes all of the shared libraries and any memory mapped segments or mapped files. The Linux kernel adheres to contingencies when deciding to dump memory mappings. For instance, it will not dump memory mapped IO devices. The Virtual Dynamic Shared Object (VDSO), stack, and heap are all dumped. Shared memory mappings and anonymous mappings that have been written to are also dumped.

It is true that ELF core files are extremely useful when debugging, especially when there are debugging information and symbols present in the original file, but they do not lend themselves well to in-depth forensics analysis, nor were they ever intended to serve as such. Also they are not very useful as a standalone file; you must load the original executable and the core file into GDB together in order to make real use of a core file. The *objdump* utility which usually relies strictly on ELF section headers will actually allow you to read a core file even in the absence of section headers but only in the simplest way (using phdr's), which would be nearly useless for forensics. The purpose of this paper is *not* to present a new debugging format. I show how the ELF core file format can be extended for the use of taking usable process memory snapshots for the purpose of forensics analysis. The ECFS format is almost a hybrid of the original executable file and a core file merged together, as it contains many of the niceties found in executables such as symbol tables and section headers.

2.2 What do I mean by forensic analysis?

Forensic Analysts and Reverse Engineers use techniques to detect, analyze, collect, and save evidence from computing devices that can be used in a court of law or for company or research interest. They apply the techniques to perform computer forensics.

In this paper I consider forensic analysis of ELF runtime infections similar to infections shown in my 2009 research [7] and [8], including any modifications to the memory image that reflects Virus, rootkit, backdoor, or exploitation remnants. To further refine scope, the following key components are isolated:

- PLT/GOT hooks
- Function trampolines
- .ctors/.dtors
- Shared library injection
- Code modifications
- ET_REL injection

- Executable anonymous mappings
- Strange segment permissions
- Suspicious threads

Currently, it is possible to analyze a live process or a core dump to inspect all of these key components if you know where to look. In fact, VMA Vudu [1] is specifically designed with the capability to do just that. However, what about those of us who don't have special software that knows how to parse and analyze the complex intricacies of the ELF format and the layout of VMA's (Virtual Memory Areas) in a process? This is exactly why I designed ECFS, my new implementation of the VMA Vudu [1] snapshot format. With ECFS format, software does not have to work so hard when analyzing a snapshot of a process. Also, without needing to manually track down common infection points and the data that is necessary to see integrity deviations, the snapshot file can be analyzed with existing *binutils* such as *objdump* and *gdb*.

3 The Extended Core File Snapshot Format

3.1 Design Goals

When designing the ECFS format, I kept several explicit goals in mind. Here is a list of my goals including an explanation of each result:

- Try to stay backwards compatible with ELF core files so that GDB can use them in the traditional way. This was somewhat challenging. So I mimicked the way that the kernel does this, and this has been mostly accomplished. However there is a problem, *objdump* will **not** look at the section headers if it sees that the file is of type ET_CORE; it automatically assumes that it should be looking at the program headers because ELF cores are not expected to have section headers whereas with ECFS Format they do. This means that it is necessary to save the snapshot as type ET_NONE instead of ET_CORE. This results in GDB not recognizing the file as a core dump, but now *objdump* can read the section headers. Whether the snapshot file is of type ET_CORE vs. ET_NONE should not matter, except in the light of using GDB. If GDB functionality is desired for loading a snapshot, one can use the --*core-format* option with the snapshot utility, and it will create a snapshot that is identical in every way with the one difference of ehdr->e_type being ET_CORE instead of ET_NONE.
- Make every component of the process image accessible via ELF section headers. This makes it possible to access important components of the process easily through parsing ELF section headers. This means that the section header table is reconstructed almost to the point of the original executable before it was loaded into memory. In addition, there are also sections created for the heap, the stack, and optionally some sections of each shared library, primarily the PLT and the PLT/GOT.

  NOTE: *In the current implementation of ECFS, the section headers are not yet created for the shared libraries and their subsections. This is currently in development.*

- Reconstruct a .dynsym symbol table but also a .symtab which contains all of the local functions. I accomplish this through the exception handling frame information in the PT_GNU_EH_FRAME segment. This feature is amazing considering the original executable is unlikely to even have a .symtab symbol table since most production applications have this symbol table stripped as it is not needed at runtime. The .dynsym table is easy to retrieve since it is locatable through the dynamic segment, but acquiring the data for reconstructing a .symtab is different and more challenging. I discuss the methods used in symbol table reconstruction in the *implementation details* part of this paper.
- Support PIE executables so that complex processes such as firefox and sshd can be properly analyzed as they would be common targets for infection. This is only slightly more challenging than a regular program and requires me to adjust offsets.

Section headers of snapshot file viewed with readelf-S:

```
$ readelf -S .snapshot/host.9691
```

```
There are 22 section headers, starting at offset 0x74000:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 1] .note             NOTE             0000000000400254  00000254
       0000000000000044  0000000000000000   A       0     0     4
  [ 2] .hash             GNU_HASH         0000000000400298  00000298
       0000000000000040  0000000000000000   A       0     0     4
  [ 3] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000040  0000000000000018   A       4     0     8
  [ 4] .dynstr           STRTAB           0000000000400360  00000360
       000000000000004f  0000000000000018   A       0     0     1
  [ 5] .rela.dyn         RELA             00000000004003e0  000003e0
       0000000000000040  0000000000000018   A       0     0     8
  [ 6] .init             PROGBITS         0000000000400488  00000488
       0000000000000040  0000000000000000  AX       0     0     8
  [ 7] .text             PROGBITS         0000000000400000  00000000
       0000000000000834  0000000000000000  AX       0     0     16
  [ 8] .fini             PROGBITS         00000000004006f4  000006f4
       0000000000000040  0000000000000000  AX       0     0     16
  [ 9] .eh_frame_hdr     PROGBITS         0000000000400708  00000708
       0000000000000034  0000000000000000  AX       0     0     16
  [10] .eh_frame         PROGBITS         000000000040073c  00000740
       000000000000073c  0000000000000000  AX       0     0     16
  [11] .dynamic          DYNAMIC          0000000000600e28  00000e28
       00000000000001d0  0000000000000008  WA       0     0     8
  [12] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000040  0000000000000008  WA       0     0     8
  [13] .data             PROGBITS         0000000000600e10  00000e10
       0000000000000468  0000000000000000  WA       0     0     8
  [14] .bss              PROGBITS         0000000000601278  00001278
       0000000000000220  0000000000000000  WA       0     0     8
  [15] .heap             PROGBITS         00000000007b8000  00000000
       0000000000021000  0000000000000000  WA       0     0     8
  [16] .stack            PROGBITS         00007fff33e3b000  001f4000
       0000000000021000  0000000000000000  WA       0     0     8
  [17] .vdso             PROGBITS         00007fff33ffe000  00212000
       0000000000002000  0000000000000000  WA       0     0     8
  [18] .vsyscall         PROGBITS         ffffffffff600000  00212000
       0000000000001000  0000000000000000  WA       0     0     8
  [19] .symtab           SYMTAB           0000000000000000  00074b34
       0000000000000078  0000000000000018          20     0     4
  [20] .strtab           STRTAB           0000000000000000  00074bac
       0000000000000037  0000000000000000           0     0     4
  [21] .shstrtab         STRTAB           0000000000000000  00074580
       00000000000000ac  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

For the first time, it is possible to view the stack and the heap using utilities such as *objdump*, not to mention the allocated .bss which is empty in the executable before runtime. It is also still possible to parse the snapshot by its program headers, as with an original core file; this is useful for obtaining segment permissions and base virtual addresses of segments, etc.

Program headers of snapshot file viewed with readelf -l:

```
$ readelf -l .snapshot/host.9691

ELF file type is NONE (None)
Entry point 0x400520
There are 23 program headers, starting at offset 476716

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                       0x0000000000000834 0x0000000000000834  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
```

```
               0x0000000000000248 0x0000000000000468  RW     200000
DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
               0x00000000000001d0 0x00000000000001d0  RW     8
NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
               0x0000000000000044 0x0000000000000044  R      4
GNU_EH_FRAME   0x0000000000000708 0x0000000000400708 0x0000000000400708
               0x0000000000000034 0x0000000000000034  R      4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000  RW     10
GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
               0x00000000000001f0 0x00000000000001f0  R      1
LOAD           0x0000000000000000 0x00000000007b8000 0x00000000007b8000
               0x0000000000021000 0x0000000000021000  RW     0
LOAD           0x0000000000000000 0x00007f6a30d13000 0x00007f6a30d13000
               0x00000000001bb000 0x00000000001bb000  R E    0
LOAD           0x0000000000000000 0x00007f6a310ce000 0x00007f6a310ce000
               0x0000000000004000 0x0000000000004000  R      0
LOAD           0x0000000000000000 0x00007f6a310d2000 0x00007f6a310d2000
               0x0000000000002000 0x0000000000002000  RW     0
LOAD           0x0000000000000000 0x00007f6a310d4000 0x00007f6a310d4000
               0x0000000000005000 0x0000000000005000  RW     0
LOAD           0x0000000000000000 0x00007f6a310d9000 0x00007f6a310d9000
               0x0000000000023000 0x0000000000023000  R E    0
LOAD           0x0000000000000000 0x00007f6a312d5000 0x00007f6a312d5000
               0x0000000000003000 0x0000000000003000  RW     0
LOAD           0x0000000000000000 0x00007f6a312f8000 0x00007f6a312f8000
               0x0000000000003000 0x0000000000003000  RW     0
LOAD           0x0000000000000000 0x00007f6a312fb000 0x00007f6a312fb000
               0x0000000000001000 0x0000000000001000  R      0
LOAD           0x0000000000000000 0x00007f6a312fc000 0x00007f6a312fc000
               0x0000000000001000 0x0000000000001000  RW     0
LOAD           0x0000000000000000 0x00007f6a312fd000 0x00007f6a312fd000
               0x0000000000001000 0x0000000000001000  RW     0
LOAD           0x0000000000000000 0x00007fff33e3b000 0x00007fff33e3b000
               0x0000000000021000 0x0000000000021000  RW     0
LOAD           0x0000000000000000 0x00007fff33ffe000 0x00007fff33ffe000
               0x0000000000002000 0x0000000000002000  R E    0
LOAD           0x0000000000000000 0xffffffffff600000 0xffffffffff600000
               0x0000000000001000 0x0000000000001000  R E    0
```

```
 Section to Segment mapping:
  Segment Sections...
   00
   01
   02     .note .hash .dynsym .dynstr .rela.dyn .init .text .fini .eh_frame_hdr
   03     .dynamic .got.plt
   04     .dynamic
   05     .note
   06     .eh_frame_hdr
   07
   08     .dynamic
   09     .heap
   10     .symtab .strtab .shstrtab
… cut …
```

3.2 Implementation Details

Up to this point, analysts had to manually locate information and reconstruct the part of the process they wanted to inspect. I perform the same, painstaking steps in my implementation of the snapshotting functionality. In this part of the paper I will explain the complexities of these procedures.

3.2.1 Section Header Table Reconstruction

The idea of reconstructing a section header table is not entirely novel. Silvio [4] shows us how to reconstruct an executable out of a core dump file and includes section headers that match up with the location of the segments. In Advances in Remote Execution Anti-forensics [5] we see how to dump a process image and reconstruct it back into an executable, also rebuilding section headers. With ECFS I followed in similar footsteps and also rebuild the section header table. However I make some progress forward into the completeness of the section headers. ECFS will reconstruct roughly 75% of the original section headers and will also include section headers for parts of the process that do not exist in the original executable including the stack, heap, vdso, vsyscall, and *soon to be,* every shared library and each shared libraries' individual sections: i.e., .libc.so.6.text, .libc.so.6.got.plt, and so forth. ECFS reconstructs several of the sections out of pure necessity, such as .eh_frame and .eh_frame_hdr, which are necessary in order for libdwarf to parse the exception handling frames. This results in a reconstructed .symtab symbol table.

The ECFS snapshot utility reconstructs each of the following section headers in the following ways:

[from program headers]

```
.interp
.note
.text
.data
```

```
.eh_frame
.eh_frame_hdr
.dynamic
.bss
```

Every process image contains the program header table for the executable within the text segment of program in memory. The program header table is located as expected by parsing the initial ELF file header which contains the offset. The program header table is parsed, and we create a section header for each corresponding segment. This means that the .text and .data sections will correspond to the text and data segments described by the phdr's, which is slightly different than the original executable's .text and .data sections. This is actually perfect, because during forensics analysis it is desirable to be able to access the entire text and data segment through reference of section headers, so tools like *objdump* can be used. The .dynamic section corresponds to the PT_DYNAMIC program header, the .interp section corresponds to the PT_INTERP program header, and so forth. The .eh_frame_hdr and .eh_frame section correspond to the PT_GNU_EH_FRAME program header and are reconstructed based primarily on that. The .bss section is created as type SHT_PROGBITS instead of SHT_NOBITS, because by the time we dump it from memory it will contain initialized static data. We are able to get its location and size by computing information from the data segment's program headers and subtracting its p_filesz from p_memsz, which always gives the expected size of the .bss area.

[From the dynamic segment]

```
.hash
.dynsym
.dynstr
.init
.fini
.got.plt
```

ECFS extracts the dynamic segment from memory, as it exists within the data segment of the executable. It utilizes the dynamic segment to locate information about where things exist that are needed by the dynamic linker including the symbol hash table, the dynamic symbol table, the global offset table, and initialization/ finalization functions. Once the dynamic segment is found, one can locate each section's corresponding dynamic tag:

```
DT_SYMTAB → .dynsym
DT_STRTAB → .dynstr
DT_HASH        → .hash
DT_INIT        → .init
DT_FINI        → .fini
DT_PLTGOT → .got.plt
```

[from the exception handling frame descriptors]

.symtab

.strtab

Once you have all of the section headers created that I described this far, you utilize the .eh_frame_hdr and .eh_frame sections. These contain the stack unwinding data that can be used for reconstructing a complete symbol table for every function that was in the original executable, which are not to be confused with the shared library functions such as *printf* which are stored in the .dynsym symbol table and trivially reconstructed from the dynamic segment. Instead, I am referring to the .symtab symbol table which will contain all of the functions that were coded into the actual executable. In previous research [6] I discussed a method for identifying functions using this same technique. With ECFS I go a step further and create an actual symbol table from the results and name each symbol like **sub_<address>.** If a function is found at 0x400440, then it is named sub_0x400440 in the symbol string table.

[from the memory mappings]

```
.heap

.stack

.vdso

.vsyscall
```

During process forensics analysis it is advantageous to access more than the program's memory segments. It is important to also access the related process segments: the heap, stack, vdso, and vsyscall page. These are all located from the /proc/<pid>/maps file, which is parsed and then translated into real memory mappings where we store the code and data of each processes' memory mappings which are eventually dumped into the snapshot file. ECFS creates program headers for each of them, including the shared libraries that are mapped into the process. The set of design goals for ECFS includes the need for creation of section headers for each of the shared libraries and their individual sections, as mentioned earlier this is not yet complete.

[finalization]

```
.shstrtab
```

Of course it is also important to add the section header that describes the string names of each section. I included this towards the end of the section header table, and it points to a string table which is near the end of the snapshot file. Then I set elfhdr->e_shstrndx to point to the section index of .shstrtab, so that it can be found.

3.2.2 Symbol Table Reconstruction

Reconstructing a symbol table from memory is something I have rarely seen done. In the case of ECFS, I reconstruct two distinct symbol tables: the dynamic symbol table which is stored in .dynsym and the local symbol table which is stored in .symtab. The latter is more challenging and potentially more useful depending on the focus of analysis. When reverse engineering, the work is clearer with a symbol table, and the same goes for process forensics analysis. Symbols allow one to make connections between code and formulate a clearer picture of how things are connecting. The dynamic symbol table contains the symbols for functions in shared libraries, such as *strcpy* and *printf*. This symbol table is relatively easy to reconstruct if you know where to look. The dynamic segment contains the location and size of the dynamic symbol table and of the dynamic symbol string table. The biggest challenge is reconstructing the .symtab symbol table which contains functions that were created in the program. Fingerprinting databases like flirt will not provide these types of functions, because they are unique from executable to executable; they will not exist in the dynamic symbol table either. In fact, most production executables are stripped and do not contain the .symtab symbol table, because it has no use except for debugging and linking purposes.

ECFS is able to reconstruct a .symtab symbol table even if it never existed in the original executable. It is able to do this, because almost every program compiled with gcc (unless it was compiled with -nostdlib) is linked with a segment of type PT_GNU_EH_FRAME which contains Frame Descriptor Entries (FDE's). FDE's are provided as stack unwinding data to be parsed by exception handling code. Even when exception handling is not being used by the code, this data is still present. The resulting snapshot file will have a .symtab symbol table that contains every function from the original executable, with the exception of one or two glibc initialization routines that are used in conjunction with the dynamic linker at initial runtime. However, those are mostly uninteresting as they are in every executable and standard. Here are the symbol tables of a simple, reconstructed program:

Readelf output showing symbol table of a snapshot file:

```
Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND pause@GLIBC_2.2.5 (2)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND malloc@GLIBC_2.2.5 (2)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND rand@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 00000000004004b0   112 FUNC    GLOBAL DEFAULT    7 sub_4004b0
     1: 0000000000400520    42 FUNC    GLOBAL DEFAULT    7 sub_400520
     2: 000000000040060d   112 FUNC    GLOBAL DEFAULT    7 sub_40060d
     3: 0000000000400680   101 FUNC    GLOBAL DEFAULT    7 sub_400680
     4: 00000000004006f0     2 FUNC    GLOBAL DEFAULT    7 sub_4006f0
```

3.3 Process Specific Information Found in /proc

The /proc file system contains a wealth of information that is stored for each process ID. For instance, the maps file contains a list of the memory mappings, and the status file contains many statistics about the process. This includes its parent, its process group, and whether or not it is being traced, etc. ECFS gets most of the process information that is needed from the /proc file system for core files when parsed by GDB. The Linux kernel performs the core dump functionality in fs/binfmt_elf.c for ET_EXEC programs and fs/binfmt_elf_fdpic.c for ET_DYN programs.

The following information is stored in ELF core files:

- process state
- register state
- auxiliary vector
- command line arguments
- shared library paths

The GDB debugger parses the program headers of a core file. It also parses the PT_NOTES segment to get the information I listed above that is stored as notes created by the kernel at dump time. The note information from the original executable is ignored, and a new notes segment is created that contains information describing the process state when it dumped. In my ECFS implementation I also ignore the original notes and convert its program header offset to point at the very end of the file, to the new note segment. The .note section header will point to the original notes from the executable, not our new notes segment. Gathering this information from /proc is relatively straight forward if you know where to look. The difficult part is setting up the notes correctly within the file. ECFS format handles this difficulty and uses the same structures from the kernel and are defined in 'sys/procfs.h'.

```
[NT_PRSTATUS]

struct elf_prstatus

  {

    struct elf_siginfo pr_info;         /* Info associated with signal.  */

    short int pr_cursig;                /* Current signal.  */

    unsigned long int pr_sigpend;       /* Set of pending signals.  */

    unsigned long int pr_sighold;       /* Set of held signals.  */

    __pid_t pr_pid;

    __pid_t pr_ppid;

    __pid_t pr_pgrp;

    __pid_t pr_sid;

    struct timeval pr_utime;            /* User time.  */

    struct timeval pr_stime;            /* System time.  */

    struct timeval pr_cutime;           /* Cumulative user time.  */

    struct timeval pr_cstime;           /* Cumulative system time.  */

    elf_gregset_t pr_reg;               /* GP registers.  */
    int pr_fpvalid;                     /* True if math copro being used.  */
  };

[NT_PRPSINFO]
struct elf_prpsinfo
  {
    char pr_state;                      /* Numeric process state.  */
    char pr_sname;                      /* Char for pr_state.  */
    char pr_zomb;                       /* Zombie.  */
    char pr_nice;                       /* Nice val.  */
    unsigned long int pr_flag;          /* Flags.  */
#if __WORDSIZE == 32
    unsigned short int pr_uid;
    unsigned short int pr_gid;
#else
    unsigned int pr_uid;
    unsigned int pr_gid;
#endif
    int pr_pid, pr_ppid, pr_pgrp, pr_sid;
    /* Lots missing */
    char pr_fname[16];                  /* Filename of executable.  */
    char pr_psargs[ELF_PRARGSZ];        /* Initial part of arg list.  */
  };
```

NT_AUXV, NT_SIGINFO, and NT_FILE also exist. Respectively each contains the auxiliary vector, the process siginfo_t struct, and a list of the memory mapped files and shared libraries.

In order to get the auxiliary vector and command line arguments one must know how the stack is laid during process load time. ECFS can get the address to the bottom of the stack where argc, argv, envp, and auxv (auxiliary vector) are all stored and uses ptrace to read them. The mapped files are of course pulled from the maps file and PTRACE_GETSIGINFO grabs the siginfo_t.

In the current prototype phase of ECFS it does not handle getting the process state from any threads of children of the process. This means that using an ECFS core file to debug a multi-threaded application will not work completely. This is another feature that is under-way.

3.4 Forensic Analysis Using the ECFS Format

I designed this new extended format, ECFS, to be easily read by simple *binutils*. Since a symbol table and section headers are included, it would behoove the analyst to use a tool such as *objdump,* which can readily read the section headers and symbol table. If an attacker has compromised a server, and one suspects that a specific process has been infected, the analyst may open the snapshot with *objdump* and look for oddities while viewing *Figure 1* (below) and checking off the list of infection types I include below.

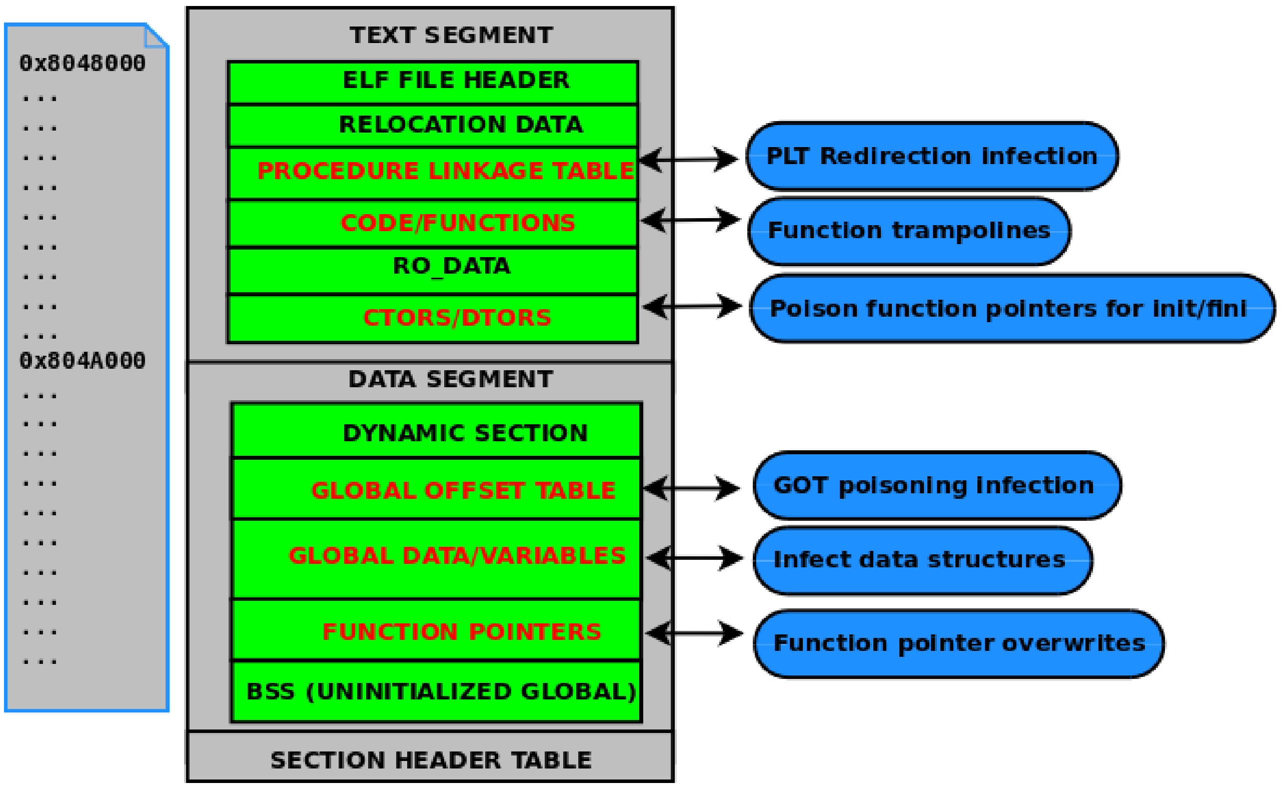

Figure 1. ELF Infection Points

3.5 Infection Types

3.5.1 PLTGOT hooks

Check each entry in the got.plt that corresponds to a shared library function and verify that each one points to an address that is within a valid shared library address space range. To determine which address spaces contain valid shared libraries the analyst must look at the DT_NEEDED entries *e.g. readelf -d* which will reveal the name of each shared library requested by the executable. I also need to note that libraries may be legitimately loaded with the *dlopen()* call. A comprehensive search through the code segment should be exhausted to determine any libraries that were loaded using this mechanism. You may then look at the address space range for each shared library as shown in the ELF notes section. If any GOT entries contain addresses that point outside of the shared libraries that were legitimately loaded via *dlopen()* or the dynamic linker (DT_NEEDED), then it can be assumed that these originate from an attackers code.

NOTE: *The first 3 entries of the GOT are always reserved and do not ever point to shared library functions.*

GOT[0] contains an address that points to the Dynamic segment of the executable, which is used by the dynamic linker for extracting useful information.

GOT[1] contains the offset into the link_map structure entry associated with the function being called/resolved.

GOT[2] contains the address to the dynamic linkers dl-resolve() function which resolves the actual symbol address for the shared library function.

3.5.2 **Function Trampolines**

Look at the prologue of each function in the text segment to see if the first 5 to 7 bytes contain any branches that redirect execution control. Remember that readelf -s <snapshot> can be used to get the location of each function.

3.5.3 **Shared Library Injection**

The path name of each shared library that was mapped into the process address space is stored in the ELF notes section, which can be viewed with *readelf -n <snapshot>*. Combine this information looking at the DT_NEEDED entries in the dynamic segment again; remember that what is listed in DT_NEEDED should be what is loaded into memory, unless there were legitimate *dlopen()* calls involved. Does *dlopen()* show up in the symbol table of the snapshot? If not, and there are discrepancies between what *readelf -n* and *readelf -d (DT_NEEDED)* are showing, then a shared library injection attack is probably in effect.

3.5.4 **Malicious Code Modifications**

Use *objcopy* to copy the .text section from the snapshot into a separate image file containing only that section; I.E *objcopy –only-section=.text <snapshot_file> <output_file>*. Then do the same thing with the original executable, copying only the .text section into another output file. So you should have something like output1 and output2. You can then *diff* these files, because they should be exactly the same. If executable code was modified in memory it should reflect here, because the two text images would differ. If they do, then you know some foul play was at work.

4 Making ECFS Snapshot Utility Available

Even though ECFS is the native snapshot format that is used by a still unreleased product, VMA Vudu, I intend to make the ECFS snapshot utility software available as its own tool for free download. My goal is to augment the use-cases for existing ELF analysis tools such as the binutils [objdump, objcopy, gdb, etc.] and to assist fellow reverse engineers and forensics analysts in their endeavors.

The ECFS snapshot utility source code is completed as a prototype, and it has to under-go so more minor development stages prior to release as a beta. With that in mind, I will have the beta version available for download on my website on January 8th, 2015.

5 References

[1] VMA Vudu – was designed in 2011 for a DARPA CFT and is currently not available publicly. For more information: http://www.bitlackeys.org/#vmavudu

[2] ELF Specification - http://www.skyfree.org/linux/references/ELF_Format.pdf

[3] The cerberus ELF interface – http://http://phrack.org/issues/61/8.html

[4] Reconstructing ELF executables from core files - http://vxheaven.org/lib/vsc03.html

[5] Advances in remote execution anti-forensics - http://phrack.org/issues/63/12.html

[6] Function identification in binaries using .eh_frame http://www.bitlackeys.org/#eh_frame

[7] Modern day ELF runtime infection via GOT poisoning http://vxheaven.org/lib/vrn00.html

[8] Saruman anti-forensics executable injector - http://www.bitlackeys.org/#saruman

[9] Volatility memory forensics tool - http://code.google.com/p/volatility/

[10] GNU binutils - http://www.gnu.org/s/binutils/

Leviathan Security Group, Inc.
3220 1st Ave S, Suite 100
Seattle, WA 98134

p: 866.452.6997
f: 206.225.2004
e: contact@leviathansecurity.com

www.leviathansecurity.com

leviathan
security group