Cheating the ELF
Subversive Dynamic Linking to Libraries

the grugq

Abstract:

Development of feature rich Unix parasites has been severely limited by the inability to reliably access functions external to the host file. Until now, it has been accepted as fact that utilizing libraries from within parasite code is a prohibitively complex task. We explore the dynamic linking mechanisms of the Executable and Linkable Format (ELF), and how these mechanisms can be bypassed or hijacked to allow parasite code access to shared objects. We demonstrate that it is not only possible, but also relatively simple, to load libraries and resolve symbols using a methodology developed within this paper. This methodology is simple to implement and can be utilized on any modern Unix supporting both the ELF and the /proc file system. Implementations of this methodology are presented for each of three popular Unix variants: Linux, FreeBSD and Solaris.

Introduction:

Recently there has been a great deal of research and active development of worms[1], both in the public realm and within the private computer underground. The virus is commonly considered a close relative of the worm. Despite this close relationship and increasing interest in worms, Unix viruses and parasites remain a poorly researched subject. There are almost no published works dealing with virus techniques on the Unix platform.

In spite of the lack of public research and documentation, Unix viruses are beginning to appear more frequently in the wild; recently Qualys Inc. discovered a virus that provides remote shell access to the infected computer[2]. Clearly underground interest in viruses is rising.

The virus is just one of a class of applications commonly referred to as parasites. For the purposes of this paper a parasite is defined as code that is injected into a host executable. The most common form of parasite is, of course, the virus; however, there are many potential uses for parasite code: binary decryption, unpacking and copyright protection, to name a few.

The parasite is an underdeveloped area of Unix security. Various constraints on the developer have hindered the development effective, complex parasites. Without a method of reliably loading libraries and resolving symbols, parasites are severely limited in their functionality. Subversive dynamic linking provides a mechanism for greatly expanding the capabilities of the Unix parasite, freeing developers from many of their previous constraints.

Dynamic linking on modern Unix platforms requires significant cooperation between the dynamic linker and the executable. The dynamic linker and executable each take

[1] Sometimes called “autonomous attack agents”.
[2] https://www.qualys.com/form_remoteshell.html

on a significant share of the tasks involved with loading libraries and resolving symbols. The executable provides a series of complex structures that the dynamic linker interprets during run time to determine which libraries to load, which symbols to resolve, etc. Adding new elements to these structures is extremely difficult after the compile time link editor has produced an executable. Due to this difficulty, virus writers have resorted to crude, unreliable methods of utilizing library functions.

The technique most frequently used involves resolving a symbol on the development machine and storing that memory address within the virus. During run time on the infected machine it is hoped that the same symbol will reside at that identical location. Clearly this is a fragile method, easily broken by only minor differences between run time environments. Additionally, this method is unable to utilize libraries not required by the host executable, and therefore not loaded as part of the process image. For instance, this limitation prevents a multi-threaded parasite from infecting a non-multi-threaded executable whose process image would not include threading libraries.

The only other technique for accessing external functions has been to utilize the existing dynamic linking framework of the host executable. This method limits the parasite to only those functions and libraries utilized by the host. This mechanism is most frequently used for interposing parasite functions in front of library functions, and almost never for acquiring functionality. There is no way for this method to work reliably across a large population of host executables, due to the large variations in library and function requirements of different executables.

The only remaining option, until now, has been to provide a copy of the required library code within the parasite itself. This is an extremely non-optimal solution for a parasite for which size is frequently a major concern. A large parasite has significantly reduced stealth capabilities and thus an increased chance of discovery.

The subversive dynamic linking methodology provides an alternative means of utilizing shared library functionality from within parasite code. This methodology enables a parasite to access functions external to the host file in a reliable manner. Developers are finally free to create complex parasites taking advantage of libraries for increased functionality.

Background

The following sections will introduce the ELF, on disk and in memory, as well as providing an overview of the dynamic linking mechanisms of the ELF.

Introducing the ELF

The Executable and Linkable Format, as the name suggests, provides two interfaces to binaries: an executable interface, and a linkable interface. The ELF header describes both interfaces, as well as basic information about the binary. The linkable interface is not required for execution; therefore, it will not be examined in this paper. The executable interface is described by program headers, which are stored in a program header table.

The program headers contain information vital to the creation of a process image, such as the location of the dynamic linking information, and how to load the file into memory. The program header table is typically located immediately after the ELF header within the binary. Each program header describes a segment, a discrete sequence of bytes, as an offset from the start of the file and a size. The type field in a program header describes how the corresponding segment should be treated, i.e. loaded into memory, interpreted as a dynamic descriptor table, etc. etc. Segments contain the program text, data, and information defining the program's run time requirements.

Segments can be loadable, in which case they describe the amount of memory that they require as well as the permissions that they expect, or they can contain information about the file. This information includes which program interpreter the executable uses, a segment describing the program header table itself, and most importantly the location of the dynamic linking descriptor table. The dynamic descriptor table is an array of simple structures that provide details of the run time environment of the ELF.

The dynamic structures each have a tag value describing how their contents should be interpreted. These contents can be interpreted as either a pointer or an integer value. The tags most important to dynamic linking all contain pointers. Pointers reference the dynamic symbol table, the dynamic string table, and the various other objects required. These other objects, which enable the transfer of control to external functions and access to external variables, will be described in the section "An introduction to ELF dynamic linking".

Process creation and runtime ELF layout

This section will explore how an ELF binary on disk is transformed into a running process in memory, and the layout of that memory image. An ELF executable is translated into a process image by the program loader. To create a process image the program loader will map all the loadable segments into memory using the `mmap()` system call, along with the loadable segments of any required libraries. After creating the process image the program loader will transfer execution control to the entry point of the primary ELF object.

Executables expect to be loaded at a fixed address[3] chosen by the link editor during compile time. ELF executables are mapped in at known memory locations, allowing the compile time linker to relocate local text and data objects. Executables need to be loaded at their chosen location in order to function correctly. Libraries might be mapped into a process image at any location; therefore, shared objects contain relocation tables to allow the dynamic linker to do last minute fix ups. Additionally, shared objects frequently contain position independent code (PIC). PIC uses local structures with the ELF image to reference external text and data objects whose location cannot be known until run time.

[3] This fixed address is usually 0x8048000 for i386 binaries, 0x10000 for 32bit SPARC v8 binaries and 0x100000000 for 64bit SPARC v9 binaries.

Loadable segments of the file are not only described by their file size, but also by the size of the memory segment that they will occupy. This run time size must be rounded up to the nearest memory page. Since most loadable segments are not exact multiples of the page size, they will be padded out in memory. The padding content is the surrounding portions of the file. This padding preserves the headers, both ELF and program, at the base of the first memory segment. This preservation allows the memory image to be interpreted as an ELF object. The ability to examine a process image as a collection of ELF objects is what enables traditional, and subversive, dynamic linking.

An introduction to ELF dynamic linking

Successful creation and execution of a process image requires more than simply memory layout information. Provisions for referencing objects whose absolute addresses are not known to the compile time link editor are required. These provisions enable code objects (functions) and data objects (`extern` variables) to be referenced between ELF memory maps within a process image. This referencing is, of course, dynamic linking. The runtime link editor (`rtld`) provides dynamic linking functionality, loading shared objects and resolving symbol references. Frequently installed as `ld.so` the dynamic linker might be either a shared object itself, or an executable.

Symbol resolution during run time is a complex and elaborate process involving significant co-operation between the executable, the libraries and the dynamic linker. The mechanisms used are unique to each of our target architectures; however, both i386 and SPARC share some common structures and methods. A description of these shared structures follows below.

Each object made available to another ELF is described by a symbol entry within the symbol table. A symbol entry is in fact a symbol structure detailing the name of the symbol, and providing a value for the symbol. The symbol name is encoded as an index into the dynamic string table. The value of a symbol is the address of that symbol within the ELF object. This address usually needs to be relocated with the base load address of the object to determine the absolute memory address of the symbol. Executables know what their load address will be during runtime and so their internally referencing symbols are relocated at compile time.

The global offset table (GOT) is an array, located within the data segment of an ELF image, which contains pointers to objects, generally data objects. The dynamic linker will fix up GOT entries, for which it has symbol entries, while loading the data segment. To access a variable whose location is not known during compilation the ELF can dereference pointers contained within the local GOT. The GOT also plays an important role in i386 dynamic linking.

The procedure linkage table (PLT) is a structure whose entries contain code fragments that transfer control to external procedures. The PLT and its code fragment entries have the following format on the i386 architecture:

```
PLT0:
            push  GOT[1]       ; word of identifying information
            jmp   GOT[2]       ; pointer to rtld function
            nop
            ...
PLTn:       jmp   GOT[x + n]   ; GOT offset of symbol address
            push  n            ; relocation offset of symbol
            jmp   PLT0         ; call the rtld

PLTn + 1    jmp   GOT[x +n +1]; GOT offset of symbol address
            push  n +1         ; relocation offset of symbol
            jmp   PLT0         ; call the rtld
```

When an executable transfers control to an external function, it passes execution to the PLT entry set up for that symbol by the compile time link editor. The first instruction in that PLT entry will jump to a pointer stored in the GOT; which, if the symbol hasn't been resolved, will contain the address of the next instruction within the PLT entry. This instruction pushes an offset in the relocation table onto the stack, and the next instruction passes execution to the zero entry in the PLT. The zero entry contains code that calls the run time link editor's symbol resolution function. This is achieved using the address of a function within the dynamic link editor, inserted into the second GOT entry by the program loader.

The dynamic linker will unwind the stack and retrieve the information needed to locate the relocation table entry. The relocation entry is used, in conjunction with the symbol and string tables, to determine which symbol the PLT entry refers to, and where that symbol's address should be stored in private memory. This symbol is resolved, if possible, and the address located is stored in the GOT entry used by the PLT entry. The next time the symbol is requested the GOT pointer will contain the address of the symbol. Thus all subsequent calls will transfer control via the GOT. The dynamic linker only resolves a symbol when it is first referenced by the binary; this is referred to as lazy loading. This lazy loading methodology of symbol resolution is the default for all ELF implementations.

In addition to the symbol table, the global offset table, the procedure linkage table, and the string table, ELF objects also contain a hash table and chain to make resolving symbols easier for the dynamic linker. The hash table and the chain, is used to rapidly determine which entries in the symbol table might correspond to a requested symbol name. This hash table is stored, along with an accompanying chain, as an array of integers. The hash table reserves the first two positions for a count of the buckets within the hash table, and a count of the elements in the chain, respectively. The hash table itself directly mirrors the symbol table both in the number of elements and in their order.

The dynamic linking structures provide all dynamically linked executables with implicit access to the dynamic linker; however, explicit access is also available. Dynamic linking, the loading of shared objects and the resolution of symbols, can be accomplished via directly accessing the run time link editor with the functions: **`dlopen()`**, **`dlsym()`** and **`dlclose()`**. These functions are contained within the

dynamic linker itself. The dynamic linking library (`libdl`) needs to be linked into the executable in order to access these functions. This library contains stub functions to allow the compile time link editor to resolve the function references; however these stub functions simply return zero. Because the actual functionality resides within the dynamic linker, shared object loading will fail if called from a statically linked ELF binary.

The information required to implement dynamic linking is: the hash table, the number of hash table elements, the chain, the dynamic string table and the dynamic symbol table. Given this information, the following algorithm will provide the address of any symbol:

```
1. hn = elf_hash(sym_name) % nbuckets;

2. for (ndx = hash[ hn ]; ndx; ndx = chain[ ndx ]) {
3.     symbol = sym_tab + ndx;

4.     if (strcmp(sym_name, str_tab + symbol->st_name) == 0)
5.           return (load_addr + symbol->st_value);
  }
```

The hash number is computed from the value of the return of `elf_hash()`, defined in the ELF specification[4], modulo the number of elements in the hash table (line 1). This number is used to reference into the hash table and discover the index of the first of the chain of symbols whose names match that hash value (line 2). Using this index, the symbol is retrieved from the symbol table (line 3). The requested symbol name is compared against the name of the retrieved symbol (line 4). If there is a match, then the location of the symbol, appended to the load address of the object, is the address of the requested symbol (line 5). If, however, there is not a match, then the chain is followed until there are no more index values (line 2). Additional checks for symbol type, i.e. data object vs. code object, as well as error checking have been left out for the sake of clarity. Using this algorithm, it is a simple matter to resolve arbitrary symbols to absolute locations in memory.

Examining Processes via procfs

Modern Unix systems provide two different methods for examining a process[5]. The POSIX standard compliant method of process inspection is the `ptrace()` system call, which provides crude, very limited, access to the memory image of a process. A far superior process examination mechanism is the proc file system, commonly called procfs, or `/proc`. This file system is typically available on all modern Unix systems.

The proc file system provides access to a process via existing file system primitives (`open()`, `read()`, etc. etc.) allowing any application to easily examine the state of an

[4] The Intel Corporation, "Tools Interface Standards: Portable Formats Specification Version 1.1 Vol 1, ELF:Executable and Linkable Format" pg 2-19

[5] We differentiate between process examination, looking at process memory, and debugging, manipulating the process under inspection.

arbitrary process. Under the procfs directory **`/proc`** each process on the system has a directory. Each directory name is the process identification (**`pid`**) of a process. Additionally, there is usually a special directory, **`self`**, that is a symbolic link to the current process' directory entry. Thus a process can examine itself using **`/proc/self`** to locate the procfs information.

The exact layout and format for procfs files is operating system dependant; however, there is typically a file describing the current state of the process (i.e. running, waiting, zombie); another file that corresponds to the binary used to create the process image; a file which gives access to the address space of the process and, most important to subversive linking, a file describing the memory maps of the process image.

Subversive Dynamic Linking Theory

Subversive dynamic linking is not based on loading libraries, but rather on locating existing procedures that perform this function. The dynamic link editor has to contain functions providing library loading. It is simply a matter of locating and utilizing these functions. The methodology is therefore:

1) **Locate the ELF object providing the library loading functions**
2) **Locate the functions that load and unload shared objects**
    a) **(Optional) locate the function that resolves symbols**
3) **Provide shared object loading, unloading and symbol resolution**

The first step is the most complex and difficult. The parasite must examine the process image of its host and discover which memory map corresponds to the required object. This is made possible with the aid of the proc file system, and some intimate ELF knowledge[6]. Parsing the procfs "maps" file is easily accomplished with only a few helper functions. The challenge is to determine which of the many memory maps that constitute a process image corresponds to the correct ELF object. The mechanism of determining which memory map is the run time dynamic linker is unique within each of the implementations, and thus will be described within the appropriate section.

The second step, locating the shared object loading and unloading functions, requires resolving the symbols **`dlopen()`** and **`dlclose()`** into absolute addresses within the ELF object. Resolving a symbol within an ELF object is quite simple, provided that there is access to the dynamic linking information: namely, the hash table, the symbol table and the string table. This information can easily be extracted from the dynamic segment, which, in turn, can be easily found using the ELF header located at the base address of the memory map. Thus, resolving the necessary symbols requires only the base load address of the target object.

[6] While possible to do without the procfs, it is significantly more reliable with this process examination aide.

The same object that manages loading libraries usually also resolves symbols within those libraries. This is typically the case, allowing the parasite to resolve the symbol `dlsym()` within the same memory map as the other dynamic linking functions. When the object that loads libraries doesn't resolve symbols, two options present themselves: either locate the `dlsym()` function within another ELF object, or provide symbol resolution functionality within the parasite code. The subversive linking implementation utilizes the second option because code to resolve symbols must already exist in order to locate the initial library loading functions.

The third, and final, step involves managing the information obtained during the first two steps, as well as any additional information gathered during run time. This data management can be accomplished in numerous ways. The mechanism chosen for the current implementations of the subversive linking methodology is a linked list whose nodes are stored on the heap. Storing data on the heap allows persistence throughout the execution lifetime of the parasite, as well as dynamic memory management.

The primary purpose of subversive linking is to provide access to the library loading functions, and to resolve symbols. This is accomplished through function pointers to the `dlopen()` and `dlsym()` procedures. These pointers, as well as pointers to the heap management functions `malloc()` and `free()`, can be stored in a structure. An opaque pointer to this structure can be managed by the parasite, which then passes it to each subversive dynamic linking function call. The pointers returned by the `dlopen()` function can be stored in a linked list, attached to the initial management structure, as well as returned to the parasite for later calls to `dlsym()`. Garbage collection is then a simple matter of traversing the linked list and `dlclose()`ing each loaded library, and `free()`ing the list node. The host's process image can thus be returned to its original pristine state in a painless and simple manner.

Implementation details

Having explained the theory we turn now to the practice of subversive dynamic linking. The following sections will examine the specifics of implementing the subversive dynamic linking methodology on each of three Unix platforms: Linux, FreeBSD and Solaris.

Linux

The first step of the methodology requires knowing which ELF object provides shared object loading functionality. Under Linux, the object that provides this functionality is the GNU C library (`glibc`). The actual function `dlopen()` contained within `libdl` is actually a wrapper for the `glibc` function `_dl_open()`. Therefore, the object that needs to be located for the first step of subversive dynamic linking is `glibc`.

Locating the memory map that corresponds to the text segment of `glibc` involves searching the host's memory image. The file `/proc/self/maps` provides access to the memory maps of the process. This file is comprised of ASCII strings, having the format:

```
/* addr range    prot  offset  dev   inode         path name    */
4001b000-400ff00 r-xp 00000000 03:01 390597     /lib/libc-2.1.3.so
```

The first field is the base load address of the object, followed by the upper limit of the memory map. This field is followed by a description of the protection on that map, read, write, execute and private (copy on write), represented by **r w x** and **p**, respectively. The next three fields are meaningless to subversive linking, the offset, device major and minor number, and the file system inode number. The last entry is most interesting, the full path name of the source file for the object mapped into memory.

The inclusion of the path name of the object allows **glibc** to be located using only **strcmp()**. The mechanism for locating the library is thus a simple string extraction and string-searching algorithm.

```
1. for (i = 0; i < nread; i++) {

2.     start = end = buffer + i;

3.     while ((*end++ != '\n') && (*end) && (i++ < nread))
           ;

4.     *end = 0;

5.     for (ptr = end; (ptr > start) && (*ptr != ' '); ptr--)
6.         if ((*ptr == *lib_name)) &&
7.             (strncmp(ptr, lib_name, strlen(lib_name)) == 0))
8.                 return ((void *)strtol(start, NULL, 16));
  }
```

The **buffer** is a character array filled by a **read()** with the contents of the file **/proc/self/maps**. The number of bytes read is stored in **nread**. The buffer is iterated through until we run out of bytes (line 1). A standing pointer to the start of the string and a walking pointer, used to locate the end of the string, are both initialized to the current location within the **buffer** (line 2).

This algorithm is searching for strings; therefore, strings need to be extracted from an arbitrary sequence of bytes. For the purposes of this algorithm, strings are defined as sequences of bytes terminated by a new line character (**'\n'**), or an ASCII NUL (**'\0'**). Extracting a string from an arbitrary stream of bytes is made possible by searching for an end of string character, as well as error checking for the end of the **buffer** (line 3). The string is NUL terminated (line 4) to increase the speed of the **strncmp()** below.

After being extracted, a string is searched for the requested object's name. The search is accomplished by pointing to the end of the extracted string and walking backward until the first word separation character (**' '**) is found, or the start of the string is reached (line 5). As an optimization, before incurring the overhead of a function call, the first **char** of each work is compared. If these match, then a **strncmp()** is likely to be useful (line 6). The current pointer is compared against the requested name, and if there is a match the string searching is over (line 7). If the string matches the

requested library name, then the start of the string is an ASCII hexadecimal representation of the load address that needs to be converted into a pointer for later manipulation and interpretation. This conversion is done by casting the return of the function **`strtol()`** to a pointer[7] (line 8).

The **`dlsym()`** function, normally used to convert a symbolic reference into an absolute memory location, in something unique to Linux, is actually contained within **`libdl`**. This lack of ready access to **`dlsym()`** is not a big a problem. The same functions used to locate and hijack the GNU C library can be reused to locate any shared object that has been loaded into the process image.

FreeBSD

The FreeBSD implementation of the dynamic link editor provides its own ELF object loading, as well as its own symbol resolution functionality. Satisfying the first step of the methodology on FreeBSD requires locating the dynamic linker as this ELF object contains the functions **`dlopen()`**, **`dlclose()`** and **`dlsym()`**. Locating the load address of the run time linker is the first step towards resolving these symbols.

Scouring the memory maps of a process can be easily accomplished with information retrieved from the procfs. FreeBSD procfs provides a process access to information about itself via the directory: **`/proc/curproc`**. The file providing information about memory maps within the process is **`map.`** Thus, a parasite can access the memory map information of its host via **`/proc/curproc/map`**. The information is stored as a series of ASCII strings having the following format.

```
/* start  end            real      prot       priv        type */
0x804800 0x805500 13 15 0xc6e18960 r-x 21 0x0 COW NC vnode
```

The first and second fields in the string contain the start and end address of a memory range. The remaining entries store a representation of the segment's protection, along with additional information potentially of interest to a parasite; however, it is of no interest to the subversive dynamic linking methodology presented in this paper.

The mechanism used to determine which memory map corresponds to the dynamic linker requires access to the Global Offset Table (GOT). The location of the GOT is contained within the dynamic linking structures. The structure with the tag **`DT_PLTGOT`** gives the address of the GOT, allowing the dynamic linker to implement traditional ELF dynamic linking

The zero entry in the GOT array is reserved to hold the address of the dynamic linking structures; this is for the convenience of the dynamic linker itself. The first and second entries are reserved on the i386. The i386 ELF specification supplement defines the first entry, GOT[1], as a "word of identifying information". This identifying information is a pointer to the dynamic linker's private structure describing the ELF object's memory map. This information is potentially useful to a parasite in many ways, but not to this subversive linking implementation. The second entry, GOT[2], is a pointer to a function within the dynamic link editor. This function

[7] C language longs and pointers are the same size on both 32 bit (ILP32) and 64 bit (LP64) architectures.

provides the entry point into the run time linker's symbol resolution procedures. This pointer is the key to locating the dynamic linker among the many memory maps of the process image. Locating which memory map range the pointer references into makes it possible to determine the load address of the dynamic linker.

```
1. rtld_func = (char *)((int *)got)[2];

2. for (i = 0, ptr = buffer; (i < nread) && (*ptr); i++, ptr++) {

3.     start = buffer + i;

4.     load_base = (char *)strtol(start, &end, 16);

5.     if (rtld_func < load_base) {
6.           while ((*ptr++ != '\n') && (*ptr) && (i++ < nread))
                 ;
           continue;
      }

7.     load_high = (char *)strtol(++end, NULL, 16);

8.     if (rtld_func < load_high)
9.           return (load_base);
  }
```

The pointer to the run time link editor is extracted from the second entry in the global offset table (line 1). The character array **`buffer`** has been filled with the contents of **`/proc/curproc/maps`**. These contents are treated as strings: byte arrays of arbitrary lenght terminated with a new line or a NUL. Each of these strings needs to be converted into memory locations, and the **`rtld`** pointer compared against this memory range. The strings are iterated through (line 2). The pointer **`start`** is initialized to the current string location within the **`buffer`**. The start of the string is converted into a pointer by the **`strtol()`** function (line 4). If the location of the dynamic linking function is lower than the load base of the current map (line 5), then the next string is extracted and loop continues (line 6). Otherwise, the highest location of the memory map is retrieved (line 7). If the pointer is lower than the maximum range of the map (line 8), then the pointer falls within the memory map and the base load address is returned (line 9).

After locating the load address of the dynamic linker, it is merely a matter of resolving the requisite functions and managing the run time data as suggested above.

Solaris

The Solaris run time link editor provides its own **`dlopen()`**, **`dlsym()`** and **`dlclose()`**. Thus, the dynamic linker is the object that needs to be located and parsed in order to satisfy the first step of the subversive linking methodology.

The Solaris procfs file **`/proc/self/map`** is comprised of an arbitrary number of **`prmap_t`** structures. These structures, defined in **`procfs.h`**, each describe a memory map and contain, among other things, the start address and size of the memory map. A method similar to that employed under FreeBSD must be used to determine which memory map corresponds to the dynamic linker. A pointer to a run time link editor function is located and the memory maps searched for the correct address range.

The SPARC Procedure Linkage Table (PLT) is stored in private memory as the link editor directly modifies it. The GOT plays no part in symbol resolution. The first four PLT entries are reserved for the dynamic link editor. The zero entry contains a small stub function to call the run time linker's symbol resolution routines, and the first entry contains a "word of identifying information". The second and third entries are used for 64bit programs, but the logic employed remains the same. The stub function creates a register window, and then calls a dynamic linker procedure. This procedure is the entry point into the link editor's run time symbol resolution functions.

Each entry in the PLT is twelve bytes long, which allows for three SPARC opcodes. The format of SPARC opcodes requires some explanation, as it is central to the locating of the dynamic linker. All SPARC opcodes are four bytes long, and are one of four possible types; called formats, viz. format 1, format 2, etc. These formats are differentiated by the first two bits of the opcode. The **`call`** opcode, which transfers control to a location in memory, is the only format 1 instruction.

The SPARC transfer control instructions, i.e. branches and calls, are relative to the current position of the program counter, or instruction pointer. This allows the code to be easily relocated in memory without relocation fix ups. The **`call`** opcode adds a signed 32bit displacement to the program counter, and thus transfers control. Since all SPARC control transfer operations are aligned on four byte boundaries, because all opcodes are four bytes long, the lower two bits of the displacement will be zero. The 32bit displacement can thus be stored as a 30bit number, allowing the most significant two bits to be recovered to indicate the opcode type.

Recovering the pointer is possible by extracting the information from the **`call`** opcode that actually transfers control to the dynamic linker. The second instruction (1-indexed, i.e. the 4^{th} byte) in the zero entry of the procedure linkage table (PLT0) is the relevant **`call`** instruction. Extracting this pointer requires reclaiming the 32bit offset, accomplished by shifting the opcode left two places; this discounts the opcode type information stored in the most significant two bits. The offset can then be added to the address of the call instruction (PLT0 + 4), which would be the value of the program counter under normal circumstances, to locate the absolute memory location of the dynamic linker's symbol resolution function. This absolute memory location can then be checked to see which memory map address range it falls into, and the base load address of the dynamic linker can thus be determined.

```
1. rtld_func = (char *) &plt[1];
2. rtld_func += (char *) (plt[1] << 2);

3. for (prmap = buf; prmap < buf + sizeof buf; prmap++)
4.     if ((rtld_func > (char *)prmap->pr_vaddr) &&
5.           (rtld_func < (char *)(prmap->pr_vaddr + prmap->pr_size)))
6.                 return (prmap->pr_vaddr);
```

The address of the **call** instruction in the zero entry is stored in **rtld_func**, which then acts as a pseudo-program counter (line 1). The 32bit displacement of the **call** instruction is then extracted and added to the pseudo-program counter to locate the absolute memory address of the dynamic link's function (line 2). The byte array **buf** is filled with the contents of **/proc/self/map**: an array of **prmap_t** structures. This array of structures is iterated through (line 3), and each structure's address range is checked to see if it includes the address of the rtld function. First, the base address of the memory map is checked, if the rtld function is lower than the base address, then the next structure is checked (line 4). If the location of the function is higher than the base address, and lower than the highest address (line 5), then the base address of the map is the load address of the dynamic linker. This load address is returned (line 6).

After locating the load address of the dynamic linker, it is merely a matter of resolving the requisite functions and managing the run time data as suggested above.

Conclusion

We have demonstrated the importance of dynamic linking for parasite code, and how a reliable mechanism of utilizing libraries greatly enhances parasite functionality. The methodology provided involved using platform specific algorithms to determine the memory address of the **dlopen()** function within the already loaded dynamic linker. With sample implementations for Linux, FreeBSD and Solaris we have demonstrated that this method is legitimate and portable. Thus, the threat of under-featured parasites has been ended forever.

Appendices
Appendix A: ELF Headers

The format of an ELF header is as follows:

```
#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf32_Half    e_type;       /* Object file type */
  Elf32_Half    e_machine;    /* Architecture */
  Elf32_Word    e_version;    /* Object file version */
  Elf32_Addr    e_entry;      /* Entry point virtual address */
  Elf32_Off     e_phoff;      /* Program header table file offset */
  Elf32_Off     e_shoff;      /* Section header table file offset */
```

```
  Elf32_Word    e_flags;      /* Processor-specific flags */
  Elf32_Half    e_ehsize;     /* ELF header size in bytes */
  Elf32_Half    e_phentsize;  /* Program header table entry size */
  Elf32_Half    e_phnum;      /* Program header table entry count */
  Elf32_Half    e_shentsize;  /* Section header table entry size */
  Elf32_Half    e_shnum;      /* Section header table entry count */
  Elf32_Half    e_shstrndx;   /* Section header string table index */
} Elf32_Ehdr;
```

The format of an ELF program header is as follows:

```
typedef struct
{
  Elf32_Word    p_type;                 /* Segment type */
  Elf32_Off     p_offset;               /* Segment file offset */
  Elf32_Addr    p_vaddr;                /* Segment virtual address */
  Elf32_Addr    p_paddr;                /* Segment physical address */
  Elf32_Word    p_filesz;               /* Segment size in file */
  Elf32_Word    p_memsz;                /* Segment size in memory */
  Elf32_Word    p_flags;                /* Segment flags */
  Elf32_Word    p_align;                /* Segment alignment */
} Elf32_Phdr;
```

The format of an ELF dynamic linking structure is as follows:

```
typedef struct
{
  Elf32_Sword   d_tag;                    /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;                   /* Integer value */
      Elf32_Addr d_ptr;                   /* Address value */
    } d_un;
} Elf32_Dyn;
```

Appendix B: Generic ELF image parser

```
ehdr = load_addr;
phdr = load_addr + ehdr->e_phoff

for (i = 0; i < ehdr->e_phnum; i++, phdr++)
      if (phdr->p_type == PT_DYNAMIC)
            break;

dyn = load_addr + phdr->p_vaddr;

for (; dyn->d_tag; dyn++) {
      switch(dyn->d_tag) {
      case DT_STRTAB:
            str_tab = load_addr + dyn->d_ptr;
            break;
      case DT_SYMTAB:
            sym_tab = load_addr + dyn->d_ptr;
            break;
      case DT_HASH:
            {
```

```
                    Elf32_Word  *p;

                    p = load_addr + dyn->d_ptr;

                    nbuckets = *p++;
                    nchains = *p++;
                    hash = p;
                    chain = p + nbuckets;
              }
              break;
        default:
              break;
        }
}
```

Appendix C: Generic dynamic linker locator

```
ehdr = BASE_ADDR;
phdr = ehdr + ehdr->e_phoff;

for (i = 0; i < ehdr->e_phnum; i++, phdr++)
      if (phdr->p_type == PT_DYNAMIC)
            break;

dyn = phdr->p_vaddr;

for (; dyn->d_tag; dyn++)
      if (dyn->d_tag == DT_PLTGOT)
            pltgot = dyn->d_ptr;

load_addr = locate_rtld(pltgot);
```

Bibliography

1. TIS Committee. “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2”, TIS Committee, 1995.
2. Paul, Richard P "SPARC Architecure, Assembly Language Programming, and C". Prentice Hall, Upper Saddle River, NJ, 2000

Acknowledgements

To the following people I owe a great debt:
That awesome bastard mammon_ for wasting years of his life at university so I wouldn’t have to; Doc Marvel, for encouraging me to waste years of my life at university (but not giving me the money for it); Silvio Cesare, for ELF conversations that would bore normal men to death; Grendel, PhD, for spending his life at university; _dose, for no real reason; and last, but not least, Knotty Dread for living the university life without me (bastard).