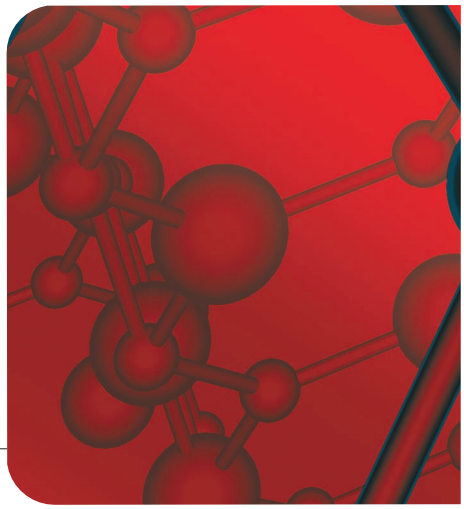
Taking a Lesson from Stealthy Rootkits



Attackers use rootkits and obfuscation techniques to hide while covertly extracting information from commercial applications. The authors describe how developers can use similar obfuscation approaches to build more agile, less-vulnerable software.

Sandra Ring and Eric Cole
The Sytex Group

After breaking into a system, attackers usually install *rootkits* to create secret backdoors and cover their tracks. Unlike the name implies, rootkits don't provide root access. Instead, they arm attackers with stealth on already compromised systems. Stealthy operations hide processes, files, and connections that let an attacker sustain long-term access without alerting system administrators. (See the "Rootkit 101" sidebar for more details on rootkits.)

Fortunately, most rootkits suffer from a lack of covertness and secrecy within their binaries. This lets administrators with access to the binary, or kernel, memory, analyze it for suspicious string and symbol characteristics. They can extract the strings and symbols and determine what attackers are doing to their systems. Unfortunately, attackers can avoid analysis by using code-obfuscation techniques that make it difficult for system administrators to detect and analyze kernel rootkits. Merely looking at symbol-table and text-segment information, which contains function names, variables, and strings contained in a program, provides valuable insight into rootkits (and even nonmalicious programs) that do not employ obfuscation. In this article, we show how software developers can use obfuscation techniques to fight attackers who reverse-engineer or illegally distribute commercial-software.

Obfuscation strategy

Obfuscation deliberately transforms software into an identically functioning—but purposefully unreadable—form, implemented in a high-level programming language at the machine-instruction level, or, to some extent, in the compiled binary. Obfuscation's only requirement is that its generated code be functionally equivalent to its parent. As an example, we could obfuscate the

```
printk("No strings attached")
```

call to become

```
NAME((char *)decode("\x59\x78\x37\x64\
x63\x65\x7e\x79\x70\x64\x37\x76\x63\x6
3\x76\x74\x7f\x72\x73\x1d\x17")),
```

a nondescriptive function pointer that does on-the-fly interpretation of XOR'd text. XOR is a binary operator used in this example to flip bits in plaintext to create cipher text. While real encryption algorithms use the XOR operator as part of their calculations, a single-character XOR itself is not considered to be "strong" encryption without the use of nonrepeatable keys. To better protect sensitive data, the decode function in this case could be replaced with a more robust encryption/decryption algorithm such as AES (www.esat.kuleuven.ac.be/~rijmen/rijndael) or a one-time XOR pad.

Symbols and strings

Rootkit kernel modules—as well as legitimate program modules—leave fingerprints (on the disk drive and in memory) of the symbols and strings that they use. A *symbol* is a pointer to a variable or a function either in or external to a program. A *string* is a snippet of ASCII plaintext found in code between quotes. Whenever you define or use a variable or a function, you define a symbol. Whenever you place any data within quotes, such as

Rootkit 101

A rootkit is a set of software tools that lets an attacker hide processes, files, and network connections. There are two popular categories: user level and kernel level.

User-level rootkits sometimes are called Trojans because they operate by placing a Trojan horse within applications such as `ps` (reports process status), `ls` (lists directory contents), and `netstat` (prints network connections) on an exploited computer's hard drive. Popular examples of user- or application-level rootkits include Torn and Lrk5 (www.antiserver.it/backdoor-rootkit). Programs such as Tripwire (www.tripwire.com) can detect these rootkits because they operate by physically replacing or modifying files on the system's hard drive.

Kernel-level rootkits have identical capabilities, but they implant their functionality directly into a running kernel instead of corrupting disk files. Many robust and popular rootkit designs are implemented within a kernel as loadable kernel modules (LKMs; Solaris LKM rootkit, www.thc.org/papers/slkm-1.0.html). These programs inject themselves into memory via `/dev/kmem`[1] or through kernel and module static patching.[2,3] Kernel rootkits are more effective than user rootkits because they can affect the behavior of programs such as `ps`, `ls`, and `netstat` by corrupting the underlying kernel functions themselves. The applications operate correctly, but the data that they receive from the kernel has been corrupted to hide the attacker's presence. Both rootkits' behavior is identical, but kernel rootkits are more difficult to detect, which makes them more desirable to attackers.

Most rootkits are wide open to forensics analysis and reverse engineering. Each binary (especially LKMs because of their reliance on external symbols and functions) tells a story for any forensics novice who can execute `more /proc/ksyms` or `strings -a rootkit.o` on a captured rootkit. Unlike applications that can be coded and compiled to be completely autonomous, sophisticated kernel modules must reference functions and variables within the kernel; there is no such thing as static compilation for modules loaded into the kernel. In this case, the shared library required for static compilation is the kernel itself. This means that without being recompiled into the kernel, the rootkit's binary doesn't contain every function and variable that it will need to execute.

In Linux and Solaris, the executable and linkable format (ELF; http://x86.ddj.com/ftp/manuals/tools/elf.pdf) symbol table for kernel modules contains function names, their relative addresses, and function and variable names outside the code (presumably within the kernel) that are marked as undefined. When the module loads into memory and links into the kernel, the undefined symbols resolve and can then be called from within the program. The linker/loader application for each operating system is responsible for querying the symbol table of the kernel to do this resolution. On Linux, this occurs when `insmod` calls the `query_module` system call for each undefined reference. The return value is placed within the symbol table of the module. Once all of the undefined references have been resolved, the module is able to function as if it had been compiled directly into the kernel. Because of this necessary symbol resolution process, variable and function names within the module binary must remain until the module is linked and loaded into memory. Therefore, the concept of strip (a GNU utility in Binutils, a collection of binary tools, which discards symbols from object files; www.gnu.org) to remove all symbols, doesn't exist (or rather it does exist, but your program will cease to function if you use it). If you strip the external symbol names from the binary prior to resolution, the program won't know the addresses where the variable or functions reside. However, we will demonstrate techniques that developers can use to remove these strings and symbols without preventing symbol resolution.

For more details on rootkit operation, see *Exploiting Software: How to Break Code*.[4]

References

1. S. Cesare, "Runtime Kernel Kmem Patching," Nov. 1998; http://vx.netlux.org/lib/vsc07.html.
2. Jbtzhm, "Static Kernel Patching," *Phrack*, vol. 0x0b, no. 0x3c, 2002; www.phrack.org/phrack/60/p60-0x08.txt.
3. Truff, "Infecting Loadable Kernel Modules," *Phrack*, vol. 0x0b, no. 0x3d, 2003; www.phrack.org/phrack/61/p61-0x0a_Infecting_Loadable_Kernel_Modules.txt.
4. G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.

assigning a variable to "my string" or printing "hello world," you create a string. Figure 1 illustrates the differences between strings, local symbols, and global symbols found in a kernel rootkit's binary.

In this example, the rootkit is implemented as a loadable kernel module (LKM), which means that this information remains present in memory once the module loads. This same kind of information exists in the binaries and memory of legitimate modules that valid users started. Attackers can leverage this information to reverse engineer or modify legitimate commercial or privately developed modules with relative ease when these sensitive strings and symbols aren't protected from sight.

Development and debugging tools, such as hexedit (a hexadecimal file viewer and editor; http://merd.sourceforge.net/pixel/hexedit.html), GDB (a GNU source-level debugger for C, C++ and other languages; www.gnu.org/directory/devel/debug/gdb.html), and Binutils (a GNU collection of binary utilities, http://sources.redhat.com/binutils/), can help gather symbol table information.

How much obfuscation?

As with any computer security–related technology, the

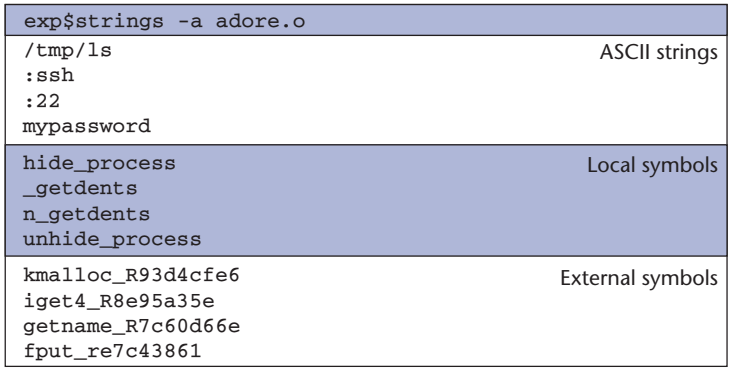


Figure 1. A rootkit's fingerprints. This sampling from a kernel rootkit identifies the location of a hidden file (`/tmp/ls`), pattern of hidden ports (:ssh and :22), password of the rootkit (mypassword), names of local functions (`hide_process`, and so on), and names of called kernel functions (`kmalloc`, and so on).

```
#!/usr/bin/perl
$BASE = "OxO";
$jump = 0;

open(DEFINES, '>>./defines.h');
foreach $file (@ARGV) {
    open(SYMBOLS, "nm -p $file |");
    $date = `date`;
    print DEFINES "// Automagically generated by
       defsym.pl \n";
      print DEFINES "// $file $date";
    while(<SYMBOLS>) {
        ($value,$type,$symbol) = split(/[ \t]+/);
        if (($type eq "t") && !
           ($symbol=~/__(.*)/)) {
           chop($symbol);
           $newsymbol = $BASE.$jump++;
      print "Replacing local symbol $symbol with
         $newsymbol\n";
           print DEFINES "#define $symbol
              $newsymbol\n";
    }
  }
    print DEFINES "\n\n";
    close(SYMBOLS);
}
```

Figure 2. A Perl script. It uses the nm symbol table display tool to determine which symbols are local and, subsequently, to build a list of `#define` replacement declarations.

challenge is to strike a balance between usability and security. We easily can obfuscate software to the point of uselessness by increasing its complexity until the efficiency and performance measurably degrade. While other more CPU-intensive methods exist, we limit discussion of obfuscation in this article to the relatively efficient removal of strings and symbols within a binary. Techniques we'll demonstrate include removing all ASCII plaintext strings, renaming local symbols, and resolving dynamic symbols for external variables and functions.

The end result is a binary that contains no readable strings and only the symbols necessary to populate an internal symbol table with their true functionalities. Because these techniques are limited to strings and symbols and don't introduce additional false execution paths, performance impact is minimal. In addition, these techniques are relatively simple to implement, requiring little overhead to integrate into most efforts. They might not prevent a fanatical attacker from reverse engineering software, but they will dramatically raise the bar of complexity and reduce the likelihood that less-dedicated attackers will succeed.

ASCII plaintext string-removal technique

Generally, there are two separate circumstances in which strings are present in binaries. The first occurs when a string is directly assigned to a variable, such as in the storage of the files and ports illustrated in Figure 1. We usually find these in two forms:

```
#define SECRET_FILE "/tmp/ls"
```

or

```
char secret_file[] = "/tmp/ls";.
```

Similarly, we might find strings embedded within functions, such as

```
if (strstr(file,"/tmp/ls") == 0).
```

In this case, the string never gets assigned to a variable but it's still present in the binary.

The easiest method to remove ASCII strings from a binary is to encrypt or encode them. A simple character-by-character XOR can obfuscate the strings to make them unreadable:

```
char secret_file[] = "\x38\x63\x7a\x67\
x38\x7b\x64";
```

and

```
if (strstr(file,(char *)decode("\x38\
x63\x7a\x67\x38\x7b\x64",23,0,7)) == 0).
```

The first example is a standard character-string assignment to the encoded equivalent. The second changes the string into a function pointer that returns the decoded string as its result. The user observes no difference functionally, but the binary is more secure because it reduces the amount of sensitive information that's available. The earlier example uses the one-byte key 0x23 across the entire string. While not considered strong in its present state, apparent in repetitive cipher text indicating the location of "`/`" earlier, it could be strengthened by increasing the key length to prevent repeatable values or by adopting a more robust algorithm.

Protecting local symbols

Local symbols refer to the items that are physically located within the kernel module (that is, the `hide_process` function in Figure 1). In general, local symbols aren't necessary within a binary because relative addressing can reference them. Their addresses are known and, unlike external references, no resolution is necessary. You can minimize the number of local symbols by avoiding global declarations and declaring as many inline functions as possible. You can rename remaining symbols using manual or automated `#define` replacement. For example, you could replace the `hide_process` symbol in Figure 1 with

```
#define hide_process "ABC".
```

ABC will replace the `hide_process` symbol everywhere it occurs within the binary. The beauty of a `#define` is that you don't have to modify any source code, so developers still can read it. Figure 2 contains a Perl script that generates a list of `#define` replacements based on the nm symbol-table display tool's output. (The nm tool is a GNU utility that displays symbols in object files; it's part of the Binutils package.)

Using nm, Figure 2's script categorizes an object's symbols as `local` or `external`. The script generates a replacement `#define` for each local symbol. Then, the program compiles a second time to utilize the new `#define` declarations. The process removes any references to the original local symbol names in the binary.

Resolving external symbols

The primary goal of eliminating external symbols from a binary is to remove references to functions that might be considered alerting or suggestive of the techniques employed within the module. External symbols are variables and functions not present in the binary but which are required for it to function properly. For example, an LKM might use `printk` but the function that implements it resides elsewhere in the kernel. LKMs aren't autonomous though; they must interact with the kernel. Figure 3 shows two paths that source code can use to reach the kernel.

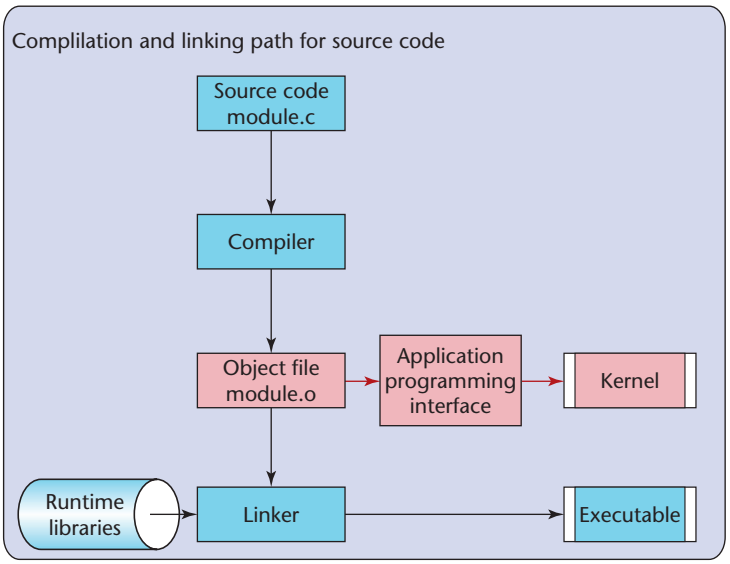


Figure 3. The source code's two paths of execution are traveling from an object file into the kernel through specially designed application interfaces or direct injections into the kernel as an executable program.

Both cases start by compiling the source into an object file identified by a .o file extension. Object code is a binary version of the source code and must be linked in with other object files. For LKMs, it goes directly from the object file format into the kernel using specially designed linker–loader applications such as insmod (a Linux utility for inserting modules into the kernel; www.netadmintools.com/html/insmod.man.html) and modload, a similar utility for Solaris (http://docs.sun.com). These applications allocate memory for the module and link the module into the kernel.

The alternate path is to follow the traditional step of linking against runtime libraries, essentially what happens when you recompile a kernel. Because LKMs are more popular and easier to implement than direct-injection, most rootkit code follows the first path. Both methods are functionally equivalent.

One approach references functions by addresses rather than names. A classic example of this address-to-name correspondence list is in Linux's `/boot/System.map` file. Although most users don't often do it, recompiling the kernel generates a new copy of it. Figure 4 shows an example of this file.

We also can find this address-to-name correspondence by viewing the running kernel's symbol table via `/proc/ksyms` (see Figure 5).

Address ranges start in the 0xFXXXXXXX block. These symbols correspond to LKMs, such as device drivers, which are currently loaded in memory. Following these symbols are the kernel's internal symbols, starting in

```
exp$ more /boot/System.map
c030ca6c D vm_min_readahead
c030ca70 D pagecache_lock_cacheline
c030ca70 d generic_file_vm_ops
c030ca7c D vmlist_lock
c030ca80 d slab_break_gfp_order
c030caa0 d cache_sizes
```

Figure 4. An example of symbols in Linux's /boot/System.map file. Recompiling the kernel automatically generates this file.

```
exp$ more /proc/ksyms
f89855e0 __insmod_tun_S.data_L96      [tun]
f8984060 __insmod_tun_S.text_L3488    [tun]
f8984f48 __insmod_tun_S.rodata_L20    [tun]
...
c012f2e0 get_user_pages_R30282b59
c030ca68 vm_max_readahead_Rf8c9aa3c
c030ca6c vm_min_readahead_R41ef314d
c0133170 fail_writepage_Rc4d0e111
c039c77c zone_table_Ra7268fc4
```

Figure 5. An example of exported symbols in Linux's kernel symbol table. The left column contains the address of the symbol, followed by its name. Addresses located in the 0xFXXXXXXX range belong to an LKM, and the third column in their case is the name of the module in which they're found. Addresses in the 0xCXXXXXXX range belong to the kernel.

the 0xC01XXXXX address range.

Symbols available within /proc/ksyms are restricted to those that are "exported" globally, meaning that only symbols that the kernel's developer wishes to be used are present. For example, the symbol `vmlist_lock` is in the System.map, but not in /proc/ksyms. This means the symbol is located in kernel memory, but hasn't been exported publicly for the system call `query_module` to identify. However, we still could leverage this symbol by resolving it manually. Using the entry in the System.map file, we could reference a symbol such as `vmlist_lock` by the address to the left of its name. For example, the line

```
rwlock_t **my_vmlist_lock = (rwlock
_t **)0xC030CA7C;
```

creates a variable that points to the value located at the address 0xC030CA7C. This lets a developer reference it as if it were the actual variable `vmlist_lock`. This process works for function pointers as well.

A second approach uses an exported symbol with an address near that of the private function or variable that a developer would like to utilize for this resolution. Depending on its address relative to the private symbol, for example, he or she would add or subtract bytes for resolution. In this case, we could leverage the variable `vm_min_readahead` to create a pointer to `vmlist_lock` without having to hardcode an address. We just declare the reference symbol as an external unsigned pointer, point `vmlist_lock` to it, and add 0x4 to the address because it is four addresses lower in the System.map file:

```
extern unsigned int *vm_min_readahead;
rwlock_t **my_vmlist_lock = (rwlock_t
**)&vm_min_readahead + 0x4;.
```

While more portable than the previous technique, these two approaches are time consuming and depend on the kernel's compilation. If, as in the first example, the address is hardcoded, the deployment operating system's compilation must be identical to the developer's. If, as in the second example, you leverage global symbols, then the compilation can't contain any additional symbols between the referenced symbol and the private symbol. You easily can implement these techniques on small scales but they are challenging and time consuming for large projects.

Dynamic symbol resolution

Starting with some distributions of the Linux 2.4 kernel, we can use an enhancement in the operating system to make external symbol resolution occur dynamically. On compilations that have `config_kallsyms` selected, dynamic symbol resolution is relatively easy with a function such as `resolve_symbol` based on Linux source code (see Figure 6).

Calling `resolve_symbol("vmlist_lock")` returns the address 0xC030CA7C. The Solaris function `mod_lookup` also provides a similar interface to resolve symbols from the kernel.

Dynamic resolution moves the problem of having an external symbol name in a binary to the other side of the equation by placing the name within quotes as an ASCII string, and we easily can solve this using encoding, encryption, or other translation techniques. An efficient approach is to create an internal symbol table for the binary, such as

```
char symbols[][2] = {
    {"printk",0x0},
    {"vmlist",0x0},
    {"kmalloc",0x0}
};.
```

The symbol's name is on the left, and a place for the resolved address is on the right. When you invoke the ASCII string removal techniques we previously de-

```
extern const char __start___kallsyms[];
extern const char __stop___kallsyms[];
static inline unsigned long resolve_symbol(const char *string) {
  const struct kallsyms_header *ka_hdr;
  const struct kallsyms_section *ka_sec;
  const struct kallsyms_symbol *ka_sym;
  const char *ka_str;
  int i;
  const char *p;
  if (__start___kallsyms >= __stop___kallsyms)
    return 0;
  ka_hdr = (struct kallsyms_header *)__start___kallsyms;
  ka_sec = (struct kallsyms_section *)
  ((char *)(ka_hdr) + ka_hdr->section_off);
  ka_sym = (struct kallsyms_symbol *)
  ((char *)(ka_hdr) + ka_hdr->symbol_off);
  ka_str = ((char *)(ka_hdr) + ka_hdr->string_off);
    for (i=0; i<ka_hdr->symbols; kallsyms_next_sym(ka_hdr,ka_sym),++i) {
        p = ka_str + ka_sym->name_off;
                  if (match(p,string) == 0)
            return ka_sym->symbol_addr;
    }
    return 0;
}
```

Figure 6. A Linux `kallsyms`-based symbol resolution technique that returns a symbol address for a string name passed in as a parameter. By including this function in a LKM, the linker–loader is no longer used to resolve sensitive symbols and more sophisticated obfuscation techniques can be implemented. In addition, symbols previously not exported for public use can be resolved using this method.

scribed, the symbol table becomes

```
char symbols[][5] = {
    {"\x67\x65\x7e\x79\x63\x7c",0x0},
    {"\x61\x7a\x7b\x7e\x64\x63",0x0},
    {"\x7c\x7a\x76\x7b\x7b\x78\x74",0x0}
};.
```

This function loops through the symbol table's `size` and populates the address fields by calling `resolve_symbol` for each entry. Once the address is resolved and the address stored to it, the encoded name disappears.

```
static inline int resolve_table() {
  int i,size =
     sizeof(symbols)/(sizeof(char
     *)+sizeof(ulong));
  for (i=0; i<size; i++) {
    symbols[i][1] =
       resolve_symbol(decode((char*)
       symbols[i][0];
    symbols[i][0] = NULL
    if (symbols[i][1] == 0)
      return -1;
  }
  return 1;
}
```

You must call this function prior to executing any other functions in the program that are in the internal symbol table.

Used with a decoding function, the resolution occurs on the encoded ASCII strings instead of the plaintext names. Combined, these let developers remove any reference to an external function or variable by creating a local symbol table containing the name, encoding the ASCII name with the created symbol table, and dynamically resolving the address for the symbol.

An example

We can combine the obfuscation techniques we described to build end-to-end code that is challenging to analyze, reverse engineer, or modify. As an example, we'll apply these concepts to convert the following overt line into a stealthy one:

```
printk("No strings attached");.
```

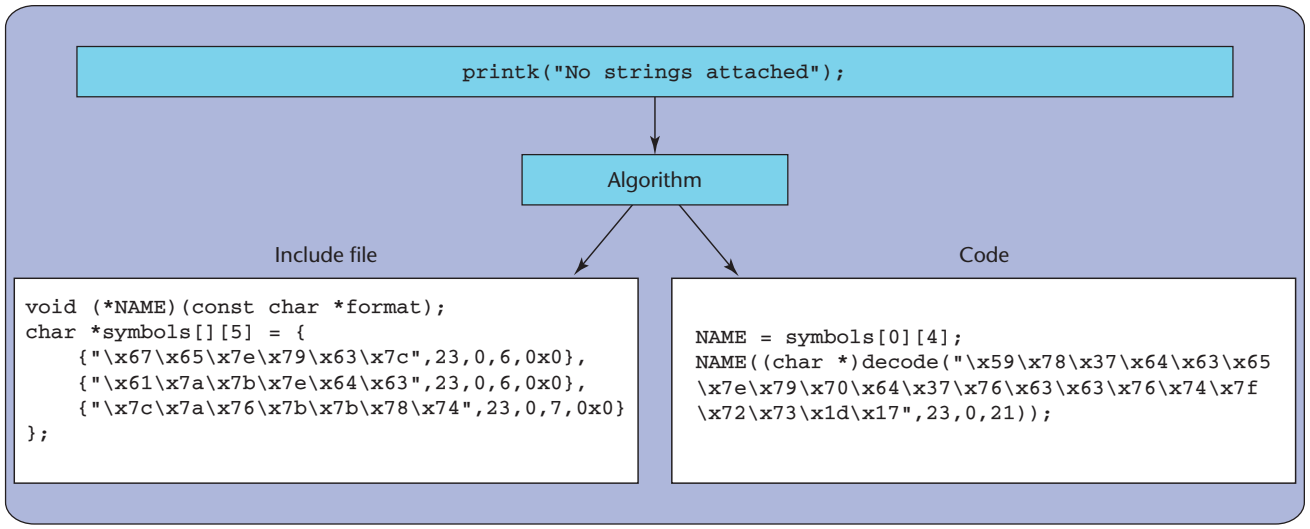


Figure 7. Conversion to stealthy code. A before-and-after view demonstrates how we can obfuscate code by using string encoding and dynamic symbol resolution.

We must remove the reference to the external function `printk` and make the ASCII string `"No strings attached"` invisible from inside the binary without modifying the program's functionality. To accomplish this, we use a program we created to parse native C code and then apply the algorithms we've already discussed. Figure 7 shows the results.

The first step in this process is defining (in an include file) an external function prototype that has a new and non-attributable symbol name that will replace it. In Figure 7, `NAME` replaces the function name `printk`. Next, we construct the symbol table, in which the first column contains the encoded value of the symbol's ASCII name. The first entry corresponds to the name `printk`, which was XOR'd with the cipher key value 0x23. The code creates a pointer to the address column for each symbol. Here, `NAME` points at the last column of the first symbol entry. The column becomes populated when the `resolve_table` function is called at the start of execution. From this point on, the function `NAME` is synonymous with the function `printk`.

Next, the text "`No strings attached`" is encoded within the function call itself. Because we want the function to operate transparently to the user, we wrap the encoded string with a call to the `decode()` function so that the plaintext string is what gets passed to the `printk` function. The result is a piece of code that operates identically to the original version, but doesn't contain the ASCII symbol `printk` or the "`No strings attached`" string.

Real-world applications

Basic obfuscation techniques, like the examples we've provided, possibly could suffer from a security-through-obscurity problem: when you're familiar with an algorithm, you know how to obfuscate (and deobfuscate) its data. We easily can overcome this problem by using robust encryption algorithms and strong encryption keys stored separately from the algorithm and cipher text. In this scheme, a key stored outside of the file (more complex then a single-character XOR value), obviates the obscurity issue because without the key, you wouldn't be able to decrypt the data even if you knew how the system worked.

Just as you shouldn't leave your house key in your home's front door lock, you also must protect the encryption key. To strengthen the system, store the key separately—not in the file itself. In addition, incorporating the computer's Internet Protocol (IP) or media access control (MAC) addresses in the obfuscation system can help prevent unwanted distribution beyond a single host.

Protecting binaries from reverse engineering, tampering, and unwanted distribution has long been a goal that many have sought.[1–3] The difficulty is that with each improvement in prevention methods comes an improvement in reverse-engineering techniques. In addition, with developers rushing to push new products to market, concepts of security and protection are often left up to the end user to implement. With the increased popularity of executable and linkable (ELF) parasitic[4] and runtime[5] viruses, mobile agents, and Unix e-commerce software, integrity will become an increasingly important factor for businesses. In the future, we hope to see this increasing importance reflected by the

incorporation of obfuscation and other antitampering techniques in product suites, and taught in colleges and universities to aspiring developers. Meanwhile, hackers will continue to use these techniques to protect their programs from analysis. The more familiar you are with these techniques, the better armed you will be to identify and take action against them when they are used with malicious intent. ❑

References

1. H. Chang and M. Atallah, "Protecting Software Code by Guards," *Workshop on Security and Privacy in Digital Rights Management 2002*, LNCS 2320, T. Sander, ed., Springer-Verlag, 2002, pp. 160–175.
2. C.S. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Trans. Software Eng.*, vol. 28, no. 8, 2002, pp. 735–746.
3. B. Horne et al., "Dynamic Self-Checking Techniques for Improved Tamper Resistance," *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, LNCS 2320, Springer-Verlag, 2001, pp. 141–159.
4. S. Cesare, "ELF Parasite and Viruses," http://packetstorm security.nl/9901-exploits/elf-pv.txt.
5. Anonymous, "Runtime Process Infection," *Phrack*, vol. 0x0b, no. 0x3b, 2002; www.phrack.org/phrack/59/p59-0x08.txt.

Sandra Ring *is deputy director of research for the Advanced Technology Research Center at The Sytex Group. Her research interests include computer and network security, autonomic computing, and forensics. She is a member of ACM and has published research on topics ranging from artificial intelligence to kernel rootkit discovery. Contact her at sring@atrc.sytexinc.com.*

Eric Cole *is chief scientist and director of research for the Advanced Technology Research Center at The Sytex Group. His research interests include steganography, cryptography, and protocol security. He has a BS and MS in computer science from New York Institute of Technology and a PhD in information technology from Pace University. Contact him at ecole@atrc.sytexinc.com.*