HTTP iframe Injecting Linux Rootkit - crowdstrike.com

By George Kurtz

Archived: 2026-04-06 01:35:10 UTC

On Tuesday, November 13, 2012, a previously unknown Linux rootkit was posted to the Full Disclosure mailing list by an anonymous victim. The rootkit was discovered on a web server that added an unknown iframe into any HTTP response sent by the web server. The victim has recovered the rootkit kernel module file and attached it to the mailing list post, asking for any information on this threat. Until today, nobody has replied on this email thread. CrowdStrike has performed a brief static analysis of the kernel module in question, and these are our results. Our results seem to be in line with Kaspersky's findings; they also already added detection.

Key Findings

- The rootkit at hand seems to be the next step in iframe injecting cyber crime operations, driving traffic to exploit kits. It could also be used in a *Waterhole* attack to conduct a targeted attack against a specific target audience without leaving much forensic trail.
- It appears that this is not a modification of a publicly available rootkit. It seems that this is contract work of an intermediate programmer with no extensive kernel experience.
- Based on the Tools, Techniques, and Procedures employed and some background information we cannot publicly disclose, a Russia-based attacker is likely.

Functional Overview

The kernel module in question has been compiled for a kernel with the version string `2.6.32-5`. The `-5` suffix is indicative of a distribution-specific kernel release. Indeed, a quick Google search reveals that the latest Debian squeeze kernel has the version number `2.6.32-5`. The module furthermore exports symbol names for all functions and global variables found in the module, apparently not declaring any private symbol as `static` in the sources. In consequence, some dead code is left within the module: the linker can't determine whether any other kernel module might want to access any of those dead-but-public functions, and subsequently it can't remove them. The module performs 6 different tasks during start-up:

1. Resolution of a series of private kernel symbols using a present `System.map` file or the kernel's run-time export of all private symbols through `/proc/kallsyms`
2. Initialization of the process and file hiding components using both inline hooks and direct kernel object manipulation
3. Creating an initial HTTP injection configuration and installing the inline function hook to hijack TCP connection contents
4. Starting a thread responsible for updating the injection configuration from a command and control server (hereafter "C2")
5. Ensuring persistence of the rootkit by making sure the kernel module is loaded at system startup
6. Hiding the kernel module itself using direct kernel object manipulation

The remainder of this blog post describes those tasks and the components they initialize in detail.

Ghetto Private Symbol Resolution

The rootkit hijacks multiple private kernel functions and global variables that don't have public and exported symbols. To obtain the private addresses of these symbols, the rootkit contains code to scan files containing a list of addresses and private symbols. Those `System.map` called files are usually installed together with a kernel image in most Linux distributions. Alternatively, the kernel exports a pseudo-file with the same syntax via procfs at `/proc/kallsyms` to userland.

The code contains the function `search_export_var` that receives one parameter: the symbol name to resolve. This function merely wraps around the sub-function `search_method_export_var` that receives an integer parameter describing the method to use for symbol resolution and the symbol name. It first attempts method `0` and then method `1` if the previous attempt failed. `search_method_export_var` then is a simple mapping of `1` to `search_method_exec_command` or `2` to `search_method_find_in_file`. Any other method input will fail. The attentive reader will notice that therefore the rootkit will always attempt to resolve symbols using `search_method_exec_command`, because method `0` is not understood by `search_method_export_var` and `2` is never supplied as input. `search_method_exec_command` uses the pseudo-file `/proc/kallsyms` to retrieve a list of all symbols. Instead of accessing these symbols directly, it creates a usermode helper process with the command line `"/bin/bash", "-c", "cat /proc/kallsyms > /.kallsyms_tmp"` to dump the symbol list into a temporary file in the root directory. It then uses a function shared with `search_method_find_in_file` to parse this text representation of addresses and symbols for the desired symbol. Due to the layout of the call graph, this will happen for every symbol to be resolved.

The alternative (but effectively dead) function `search_method_find_in_file` is, unfortunately, as ugly. Despite the fact that the System.map file is a regular file that could be read without executing a usermode helper process, the author found an ingenious way to use one anyway. Since multiple kernels might be installed on the same system, the `System.map` file(s) (generated at kernel build time) include the kernel version as a suffix. Instead of using a kernel API to determine the currently running kernel version, the rootkit starts another usermode helper process executing `"/bin/bash", "-c", "uname -r > /.kernel_version_tmp"`. `uname` is a userland helper program that displays descriptive kernel and system information. So instead of using the kernel version this module is built for at build time (it's hardcoded in other places, as we'll see later), or at least just calling the same system call that `uname` uses to obtain the kernel version, they start a userland program and redirect its output into a temporary file. The kernel version obtained in this way is then appended to the `System.map` filename so that the correct file can be opened. Recall that this code path is never taken due to a mistake at another place, though. When starting up, the rootkit first iterates over a 13-element array of fixed-length, 0-padded symbol names and resolves them using the previously described functions. The name of the symbol and its address are then inserted into a linked list. Once a symbol's address needs to be used, the code iterates over this linked list, searching for the right symbol and returning its address.

Berserk Inline Code Hooking

To hook private functions that are called without indirection (e.g., through a function pointer), the rootkit employs inline code hooking. In order to hook a function, the rootkit simply overwrites the start of the function with an

`e9` byte. This is the opcode for a `jmp rel32` instruction, which, as its only operand, has *4 bytes* relative offset to jump to. The rootkit, however, calculates an *8-byte* or 64-bit offset in a stack buffer and then copies 19 bytes (8 bytes offset, 11 bytes uninitialized) behind the e9 opcode into the target function. By pure chance the jump still works, because amd64 is a little endian architecture, so the high extra 4 bytes offset are simply ignored. To facilitate proper unhooking at unload time, the rootkit saves the original 5 bytes of function start (note that this would be the correct `jmp rel32` instruction length) into a linked list. However, since in total 19 bytes have been overwritten, unloading can't work properly:

```
.text:000000000000A32E
 xor
 eax, eax
.text:000000000000A330
 mov
 ecx, 0Ch
.text:000000000000A335
 mov
 rdi, rbx
.text:000000000000A338
 rep stosd
.text:000000000000A33A
 mov
 rsi, rbp
.text:000000000000A33D
 lea
 rdi,
.text:000000000000A341
 lea
 rdx,
.text:000000000000A345
 mov
 cl, 5
.text:000000000000A347
 rep movsd
.text:000000000000A349
 mov
 , rbp
.text:000000000000A34C
 mov
 esi, 14h
.text:000000000000A351
 mov
 rdi, rbp
.text:000000000000A354
 mov
 rax, cs:splice_func_list
.text:000000000000A35B
```

```
 mov
 , rdx
.text:000000000000A35F
 mov
 , rax
.text:000000000000A363
 mov
 qword ptr , offset splice_func_list
.text:000000000000A36B
 mov
 cs:splice_func_list, rdx
.text:000000000000A372
 call
set_addr_rw_range
.text:000000000000A377
 lea
 rax,
.text:000000000000A37B
 mov
 byte ptr , 0E9h
.text:000000000000A37F
 lea
rsi,
.text:000000000000A384
 mov
 ecx, 19
.text:000000000000A389mov
 rdi, rax
.text:000000000000A38C
 rep movsb
.text:000000000000A38E
 mov
 rdi, rax
```

To support read-only mapped code, the rootkit contains page-table manipulation code. Since the rootkit holds the global kernel lock while installing an inline hook, it could simply have abused the write-protect-enable-bit in cr0 for the sake of simplicity, though. Since the rootkit trashes the hooked function beyond repair and is not considering instruction boundaries, it can never call the original function again (a feature that most inline hooking engines normally posses). Instead, the hooked functions have all been duplicated (one function even twice) in the sourcecode of the rootkit.

File and Would-be Process Hiding

Unlike many other rootkits, this rootkit has a rather involved logic for hiding files. Most public Linux rootkits define a static secret and hide all files and directories, where this secret is part of the full file or directory name. This rootkit maintains a linked list of file or directory names to hide, and it hides them only if the containing

directory is called `".html"` or `"sound"` (the parent directory of temporary files and the module file, respectively). The actual hiding is done by inline hooking the vfs_readdir function that's called for enumerating directory contents. The replacement of that function checks if the enumerated directory's name is either `".html"` or `"sound"` as explained above. If that's the case, the function provides an alternative function pointer to the normally used `filldir` or `filldir64` functions. This alternative implementation checks the linked list of file names to hide and will remove the entry if it matches. Interestingly, it will also check a linked list of process names to hide, and it will hide the entry if it matches, too. That, however, doesn't make sense, since the actual directory name to hide would be the *process id*. Also, the parent directory for that would be `"/proc"`, which isn't one of the parent directories filtered. Therefore, the process hiding doesn't work at all:

The list of hidden files is:

- sysctl.conf
- module_init.ko (the actual rootkit filename)
- zzzzzz_write_command_in_file
- zzzzzz_command_http_inject_for_module_init

The real module's name gets added to the linked list of file names to hide by the module hiding code. Interestingly, the rootkit also contains a list of parent path names to hide files within. However, this list isn't used by the code:

- /usr/local/hide/first_hide_file
- /ah34df94987sdfgDR6JH51J9a9rh191jq97811

Since only directory listing entries are being hidden but access to those files is not intercepted, it's still possible to access the files when an absolute path is specified.

Command and Control Client

As part of module initialization, the rootkit starts a thread that connects to a single C2 server. The IP address in question is part of a range registered to Hetzner, a big German root server and co-location provider.

The rootkit uses the public ksocket library to establish TCP connections directly from the Linux kernel. After the connection has been successfully initiated, the rootkit speaks a simple custom protocol with the server. This very simple protocol consists of a 1224-byte blob sent by the rootkit to the server as an authentication secret. The blob is generated from *"encrypting"* 1224 null bytes with a 128-byte static password, the C2 address it's talking to, and, interestingly, an IP address registered to Zattoo Networks in Zurich, Switzerland, that is not otherwise used throughout the code.

The server is then expected to respond with the information about whether an iframe or a JavaScript snippet should be injected, together with the code to be injected. The server's response must contain a similarly generated authentication secret for the response to be accepted. If this check passes, the rootkit then copies the injection information into a global variable. This protocol is obviously vulnerable to simply generating the secret blob once using dynamic analysis and replaying it, and therefore it merely serves for a little obfuscation. We didn't invest further time investigating this specific *"encryption"* algorithm.

TCP Connection Hijacking

In order to actually inject the iframes (or JavaScript code references) into the HTTP traffic, the rootkit inline hooks the `tcp_sendmsg` function. This function receives one or multiple buffers to be sent out to the target and appends them to a connections outgoing buffer. The TCP code will then later retrieve data from that buffer and encapsulate it in a TCP packet for transmission. The replacement function is largely a reproduction of the original function included in the kernel sources due to the inline hooking insufficiencies explained above. A single call to the function `formation_new_tcp_msg` was added near the head of the original function; if this function returns one, the remainder of the original function is skipped and internally a replacement message is sent instead. This function always considers only the first send buffer passed, and we'll implicitly exclude all further send buffers passed to a potential `sendmsg` call in the following analysis. The `formation_new_tcp_msg` function invokes a decision function that contains 4 tests, determining whether injection on the message should be attempted at all:

1. An integer at `+0x2f0` into the current configuration is incremented. Only if its value modulo the integer at `+0x2e8` in the current configuration is equal to zero, this test passes. This ensures that only on every *n*-th send buffer an injection is attempted.
2. Ensure that the size of all the send buffers to be sent is below or equal to 19879 bytes.
3. Verify that originating port (server port for server connections) is :80.
4. Ensure that the destination of this send is not 127.0.0.1.
5. Make sure that none of the following three strings appears anywhere in the send buffer:
    - "403 Forbidden"
    - "304 Not Modified"
    - " was not found on this server."
6. Make sure the destination of this send is not in a list of 1708 blacklisted IP addresses, supposedly belonging to search engines per the symbol name `search_engines_ip_array`.

There are several shortcomings in the design of these tests that ultimately led to the discovery of this rootkit as documented in the Full Disclosure post. Since the check to only attempt an inject once every n-th send buffer is not performed per every m-th connection and before all other tests, it will trigger on more valid requests than one might expect when defining the modulus. Also, doing a negative check on a few selected error messages instead of checking for a positive "200" HTTP status led to the discovery, when an inject in a "400" HTTP error response was found. The rootkit then tries to parse a HTTP header being sent out by looking for the static header strings "Content-Type", "Content-Encoding", "Transfer-Encoding" and "Server". It matches each of the values of these headers against a list of known values, e.g., for Content-Type:

- text/html
- text/css
- application/x-javascript

The Content-Type of the response and the attacker specified Content-Type of the inject have to match for injection to continue. The code then searches for an attacker-specified substring in the message and inserts the inject after it. What is notable is the support for both `chunked` Transfer-Encoding and `gzip` Content-Encoding. The `chunked` encoding handling is limited to handling the first chunk sent because the HTTP headers parsed need to present in the same send buffer. However, it will adjust the length of the changed chunk correctly. When encountering a `gzip` Content-Encoding, the rootkit will use the zlib kernel module to decompress the response, potentially patch

it with the inject, and then recompress it. While this is a technically clever way to make sure your inject ends up in even compressed responses, it will potentially severely degrade the performance of your server.

Reboot Persistence

After running most of the other initialization tasks, the rootkit creates a kernel thread that continuously tries to modify `/etc/rc.local` to load the module at start-up. The code first tries to open the file and read it all into memory. Then it searches for the loading command in the existing file.

>If it's not found, it appends the loading command "insmod /lib/modules/2.6.32-5-amd64/kernel/sound/module_init.ko" by concatenating the "insmode" command with the directory path and filename. However, all those 3 parts are hardcoded (remember that the kernel version now hardcoded was determined dynamically for symbol resolution earlier?). If opening the file fails, the thread will wait for 5 seconds. After successfully appending the new command, the thread will wait for 3 minutes before checking for the command and potentially re-adding it again. Additionally, the rootkit installs an inline hook for the vfs_read function. If the read buffer (no matter which file it is being read from) contains the fully concatenated load command, the load command is removed from the read buffer by copying the remainder of the buffer over it and adjusting the read size accordingly. Thereby, the load command is hidden from system administrators if the rootkit is loaded.

The screenshot above showcases a problem already with this technique of persistence: since the command is appended to the end of rc.local, there might actually be shell commands that result in the command not being executed as intended. On a default Debian squeeze install, `/etc/rc.local` ends in an `exit 0` command, so that the rootkit is effectively never loaded.

Module Hiding

Hiding itself is achieved by simple direct kernel object manipulation. The rootkit iterates about the kernel linked list `modules` and removes itself from the list using `list_del`. In consequence, the module will never be unloaded and there will be no need to remove the inline hooks installed earlier. In fact, the `remove_splice_func_in_memory` function is unreferenced dead code.

Conclusion

Considering that this rootkit was used to non-selectively inject iframes into nginx webserver responses, it seems likely that this rootkit is part of a generic cyber crime operation and not a targeted attack. However, a Waterhole attack, where a site mostly visited from a certain target audience is infected, would also be plausible. Since no identifying strings yielded results in an Internet search (except for the ksocket library), it appears that this is not a modification of a publicly available rootkit. Rather, it seems that this is contract work of an intermediate programmer with no extensive kernel experience, later customized beyond repair by the buyer.

Although the code quality would be unsatisfying for a serious targeted attack, it is interesting to see the cyber-crime-oriented developers, who have partially shown great skill at developing Windows rootkits, move into the Linux rootkit direction. The lack of any obfuscation and proper HTTP response parsing, which ultimately also led to discovery of this rootkit, is a further indicator that this is not part of a sophisticated, targeted attack.

Based on the Tools, Techniques, and Procedures employed and some background information we cannot publicly disclose, a Russia-based attacker is likely. It remains an open question regarding how the attackers have gained the root privileges to install the rootkit. However, considering the code quality, a custom privilege escalation exploit seems very unlikely.
