

Dissecting a PHP web server rootkit

 sophos.com/en-us/blog/dissecting-a-php-web-server-rootkit

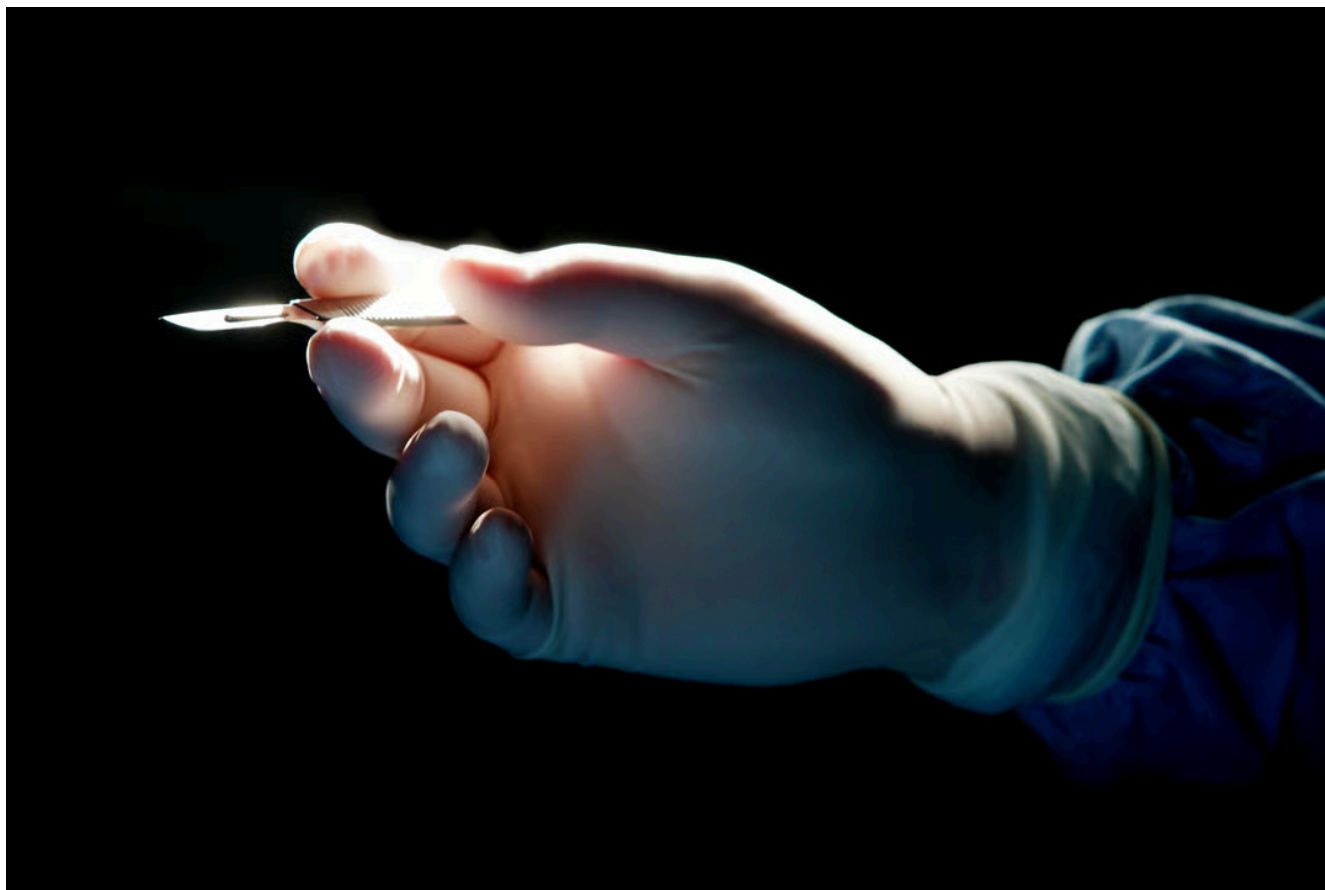
September 7, 2026

Sophos X-Ops takes a deep dive into an insidious piece of malware

September 7, 2026



Written by [Luke Mitchell](#)



SophosLabs recently acquired a Linux implant associated with compromised BIG-IP Access Policy Management (APM) environments that use Apache and PHP components. The malware demonstrates advanced techniques including custom ELF loading, function hooking, and runtime code patching to evade detection while maintaining persistent access through hidden web shells.

The implant delivers a familiar outcome – on-demand server-side code execution commonly associated with web shells – but implements it using deeper Linux- and Apache-specific tradecraft.

The malware targets deployments featuring Apache, libphp, APR module loading, BIG-IP APM webtop components, and BIG-IP upgrade workflows, suggesting it was developed for specific environments. F5 [associates](#) the related c05d5254 activity with BIG-IP APM systems affected by [CVE-2025-53521](#), an exploited unauthenticated RCE in BIG-IP APM when an access policy is configured on a virtual server. If you believe you are, or have been, using affected BIG-IP APM versions, follow F5's remediation and compromise-assessment guidance before applying generic Apache or PHP hardening recommendations.

Our analysis suggests the sample discussed here represents a second-stage payload; during parallel analysis of a related **umount** sample, we noted a distinct installer/propagation

component responsible for infecting **/usr/sbin/httpd**, persisting across BIG-IP upgrade images, modifying SELinux configurations, and deploying the payload analyzed in this article. The first-stage loader looks for BIG-IP upgrade/install-image workflows under, for example, **/mnt/tm_install**. Notably, the malicious prefix size used by the infected **httpd** (0x5430) matches the size of the payload embedded within the **umount** sample, strongly suggesting that the latter is responsible for deploying the former.

The second-stage sample hides key operational strings with RC4, gains execution before the host application `main()` function is invoked by intercepting **__libc_start_main**, targets Apache's PHP module by hooking the Apache Portable Runtime (APR) module loader (**apr_dso_load**), and injects a PHP web shell into memory. It does the latter by manipulating **mmap** behavior inside **libphp** at runtime – so that only the infected process sees the malicious content and nothing ever touches the disk.

Alongside this web-based access, the implant also creates a local UNIX domain socket and can redirect a connection into **/bin/bash**, enabling interactive access without opening a TCP listening port.

Based on current public reporting and our analysis, the observed targeting centers on BIG-IP APM webtop environments rather than generic Apache/PHP or common CMS deployments.

Note: While engaged in this research, we became aware that researchers from ESET had [conducted](#) analysis of this malware, which they dubbed 'PoisonedRefresh.' Our analysis independently observed overlapping behavior and contributes further detail.

SHA256 of sample:

26bd5b0722d1dbab5db749a063c49bc8638653ac2addfead7a9cb3d6d57bccc9

Why this case matters

When defenders hear 'web shell,' they usually think of a small server-side script, often written in PHP, JSP, or ASP, planted in a web-accessible directory to provide persistent remote code execution through ordinary HTTP requests. The script exists as a file on disk, perhaps obfuscated or hidden among legitimate content, but discoverable nonetheless.

That assumption has shaped years of detection logic. Analysts search web root directories for suspicious scripts, look for tell-tale parameter names in HTTP requests, and rely on file integrity monitoring to surface unexpected changes.

Some web shell families became notorious precisely because they demonstrated how powerful that simple attack model could be. [China Chopper](#), for example, is a well-known web

shell that MITRE describes as providing access through web servers and supporting behaviors such as HTTP POST command execution, file operations, and command-terminal access.

This sample challenges that traditional model in three ways:

- **Fileless web shell delivery:** the web shell capability still exists, but is no longer anchored to a static script file on disk. Instead, the implant intercepts the loading of specific PHP files and prepends a web shell to their in-memory representation at `mmap()` time
- **Process-level compromise, not just app-level:** the implant redirects selected **libc** and **libphp** function calls inside Apache worker processes, so every PHP-based component executing within that process - plugins, scanners, local scripts - is running inside a manipulated runtime environment
- **Dual access channels:** attackers can use both an HTTP-driven PHP payload and a local UNIX socket backdoor that gives them an interactive shell without a listening TCP port

The result is an access primitive that behaves like a web shell to the attacker, but is significantly harder to detect using file-centric or PHP-only detection alone. On a compromised host, anything executing through the tampered **libphp** runtime may observe a different view of targeted PHP files than what actually exists on disk, undermining assumptions made by traditional file-inspection tools.

While we do not have sufficient evidence to attribute this malware to a specific threat actor, the targeting and implementation suggest operational sophistication.

The web shell in a nutshell

At a conceptual level, the sample combines three ideas:

1. **Run early in process lifetime.** The implant executes before the host application's normal logic by intercepting **libc**'s startup sequence via `__libc_start_main`.
2. **Act only when PHP is present.** By hooking Apache Portable Runtime (APR) functions, notably `apr_dso_load`, the implant waits until Apache loads the php module (**libphp**) before altering process behavior.
3. **Provide multiple access paths.** A PHP web shell is injected into the in-memory representation of specific PHP files via `mmap()` interception inside **libphp**, and a local UNIX domain socket, after authentication, redirects input, output and error streams into `/bin/bash`, enabling interactive shell access without exposing a network listener.

Each of these techniques is relatively straightforward in isolation; the sophistication lies in how they are combined into a coherent, stable access mechanism targeted specifically at Apache/PHP deployments.

Our analysis

The binary was stripped and statically linked, immediately limiting the value of conventional triage approaches such as import analysis or simple string searches. The sample also bypasses the usual dynamic-linker startup path, instead bringing its own loader logic and startup hooks.

Rather than starting from imports, we focused on:

- Control-flow pivots, especially around process startup
- Runtime string decryption routines
- Symbol resolution and hook installation logic
- Use of Linux process introspection via `/proc`
- Transition points where encrypted or opaque data becomes plaintext and executable

This style of analysis is becoming increasingly important as Linux malware continues to move away from simple on-disk implants and toward custom loaders, runtime code generation, in-memory execution, and process manipulation.

While these techniques still generate observable signals defenders can use, they often replace obvious filesystem artifacts with behavioral chains that require more context, or deeper inspection to identify.

This sample is a good example: rather than dropping a conventional web shell in the usual way, it modifies how specific PHP files are presented to the running process at runtime. While the capability exposed to the attacker is similar to that of a traditional web shell, the implementation removes many of the artifacts defenders typically rely on. It highlights the importance of a defense-in-depth approach that combines filesystem, memory, process, and behavioral visibility.

RC4 strings: Following the breadcrumbs

One of the first hurdles in understanding the sample was its extensive use of runtime string decryption. Many of the most important strings are stored encrypted in the `.rodata` section and decrypted only as needed using a compact RC4 routine.

The malware uses the RC4 stream cipher with a hardcoded 16-byte key:

TrswBWII90Z5e38n.

This obscures key operational strings from simple static analysis, making it more difficult to understand the sample's functionality through conventional string extraction alone. During our analysis, we identified and decrypted a number of distinct encrypted strings, including function names, library names, and target filenames.

Once decrypted at runtime, these strings reveal the implant's true scope and intent:

- **/proc/self/exe** and **/proc/self/maps**, used for process self-inspection
- **__libc_start_main**, the **libc** startup routine
- **apr_dso_load** and **apr_time_now**, APR functions inside Apache
- **libphp**, the PHP module targeted inside the Apache process
- File and memory APIs such as **open**, **close**, and **mmap**
- Threading primitives like **pthread_create** and **pthread_detach**

Individually, none of these strings are particularly exciting, but together they paint a picture of a sample that understands Linux process internals, Apache's runtime, and PHP's execution model.

Defender tip: In this sample, RC4 appears to function primarily as an obfuscation mechanism: it hides operational strings from straightforward static extraction and delays analysis until runtime. It's worth noting here that, as with any sample, seemingly innocuous string outputs doesn't necessarily mean a binary is benign. For Linux server investigations, runtime string decryption, dynamic API resolution, and delayed revelation of configuration data are increasingly common in more capable Linux malware families.

Racing 'main' for execution

On Linux, most programs do not call **main** directly. Execution flows through **libc** initialization code which prepares the process environment and then invokes **main**. A critical component of this sequence is **__libc_start_main**, which performs initialization and calls the program's entry point.

Ordinary Linux ELF binaries follow a well-trodden path: the kernel loads the binary, the dynamic linker (**ld.so**) resolves dependencies, and then **libc's** startup code prepares the environment and calls **main**. Many security and observability tools piggy-back on this sequence to gain visibility into process startup.

Our sample deviates from this path in two ways. First, at the raw entry point, it jumps directly into a custom loader routine rather than calling **__libc_start_main** in the usual way. Rather

than simply executing additional code, the loader reopens its own image via `/proc/self/exe`, seeks to the offset where the preserved original executable begins, and manually loads that embedded ELF into memory.

```
public start
start proc near
xor     ebp, ebp
pop     esi
mov     ecx, esp
and     esp, 0FFFFFFF0h
push    eax
push    esp
push    edx
push    offset _term_proc
push    offset _init_proc
push    ecx
push    esi
push    offset loc_804E7A0
call    sub_80548F2
start endp
```



Figure 1: A typical `_start` routine from a stripped, statically linked 32-bit ELF. Even without symbols, the entry point has familiar C runtime startup behavior: stack alignment, setup of startup arguments, and a call into a `libc` startup routine. This made the infected `httpd` entry point stand out, because instead of following this pattern it passed the raw stack directly into a custom loader and halted if that routine returned.

```

public start
start proc near
mov     eax, esp
push    edx
push    eax
call    sub_20401A60
hlt
start endp

```



Figure 2: The program entry point (**`_start`**) transfers execution directly to a custom loader routine. In a conventional Linux executable, **`_start`** would normally proceed to **`__libc_start_main`**, which ultimately invokes **`main()`**. The absence of that familiar startup sequence immediately stood out during analysis and led to the discovery of the implant's custom ELF loading and startup-hooking logic.

The loader then parses the embedded ELF, walks its PT_LOAD segments, maps a new memory region, copies segment contents into place, applies the appropriate memory protections, handles the original interpreter where required, and redirects execution through its own **`__libc_start_main`** wrapper.

```

orig__libc_start_main = (int)v38 + v39; // store original
if ( dword_20406478 || *(_BYTE*)(v55 + 4) == 6 )
{
    v41 = (_WORD)v37 == 3;
    v42 = &orig__libc_start_main;
    if ( v41 )
        v42 = (int*)((char*)&orig__libc_start_main - v46);
    // Resolve __libc_start_main and preserve the original.

    // Redirect process startup through the implant wrapper,
    // ensuring malicious initialisation executes before main().
    *(_DWORD*)v55 = v42;
    *v36 = hooked__libc_start_main; // redirect libc startup to the wrapper
}
else
{
    v40 = hooked__libc_start_main;
    if ( (_WORD)v37 == 3 )
        v40 = (int)(__cdecl*)(int, int, int, int, int, int, int)((char*)hooked__libc_start_main - v46);
    *v36 = v40;
}

```



Figure 3: The implant decrypts and resolves `__libc_start_main`, preserves the original `libc` startup routine, and overwrites it with a wrapper (hooked `__libc_start_main`) so that the implant code executes before the program's real main.

In practical terms, this 'bring your own loader' approach provides the threat actor with several advantages:

- It avoids the conventional dynamic-linker startup path, potentially reducing visibility from controls that rely on that sequence as a monitoring point
- It gives the attacker precise control over when and how `libc` startup is intercepted
- It gives a clean, single pivot point into the rest of the implant

From a defender's perspective, it's another reminder that relying on the normal loader path as an interception point is no longer sufficient for more capable Linux threats.

The sample explicitly targets `__libc_start_main` and replaces it with a wrapper function. Conceptually, the flow changes from:

```
_start -> __libc_start_main(main, ...) -> main()
```

to:

```
_start -> custom loader -> real __libc_start_main(wrapper, ...) -> wrapper() -> real  
main()  
wrapper() runs implant initialization  
wrapper() then calls the real main
```

This gives the implant an early and reliable execution window before the host application performs meaningful work. It also provides a stable pivot point regardless of how the remainder of the application behaves, allowing the malware to initialize once and then blend into an otherwise legitimate process.

Intercepting startup logic in this manner also provides a threat actor with:

- A stable, single execution point
- The ability to resolve APIs and install hooks before application threads begin
- A way to make later malicious behavior appear native to the process

From a forensic perspective, it also shifts indicators earlier in the process timeline, often before logging frameworks are fully initialized.

Defender tip: Unusual behavior very early in a process's lifetime, such as access to `/proc` or memory permission changes, can be more informative than later activity.

Targeting Apache via APR

After gaining early execution, the implant does not immediately attempt to manipulate everything in the process. Instead, it waits for a specific condition: Apache loading the PHP module (**libphp**).

APR provides APIs for dynamic module loading – including **apr_dso_load**, which loads shared objects at runtime. Hooking this function gives the implant a natural observation point for module initialization, without guessing or hardcoding load order. Rather than patching every Apache process indiscriminately, the malware can wait until its intended target environment is present before activating.

Our sample hooks **apr_dso_load** and inspects module paths as they are loaded. When the malware detects **libphp**, it modifies behavior inside that module.

```
v3 = orig_apr_dso_load(a1, a2, a3);
qmemcpy(v22, &str_libphp, sizeof(v22));
rc4_decrypt_string(v22, 7);
if ( v3 )
    return v3;
// Only trigger the libphp patching path when Apache loads the PHP module.
v5 = string_length(a2);
if ( !path_contains(a2 + v5 - 16, v22) )
    return v3;
original_pid = getpid();
```

SOPHOS | X-OPS

*Figure 4: The implant hooks APR's **apr_dso_load** and explicitly gates its behavior on the PHP module, returning immediately unless **libphp** is being loaded.*

At the same time, the sample hooks **apr_time_now**, APR's 'current-time' function. This API appears innocuous, but is invoked frequently in a live Apache process.

In our assessment, the implant uses this frequently invoked API as a delayed trigger for follow-up actions, including starting the worker responsible for the local UNIX socket backdoor.

This approach suggests operational maturity in the following ways:

- The implant limits its scope to the environment it is designed for
- It avoids destabilizing services by acting too early or too broadly
- It leverages the target platform's own runtime abstractions rather than fighting them

This selective activation is a recurring theme throughout the implant. Rather than modifying the process as soon as it starts, the malware repeatedly waits for specific runtime conditions

before enabling additional functionality.

Defender tip: In Apache processes, investigate memory-protection changes or executable-page modifications occurring shortly after libphp is loaded.

Locating libphp in memory

Once PHP is loaded, the implant needs to know where it resides in process memory. Linux exposes this information via **/proc/self/maps**, which lists all memory mappings along with their address ranges and backing files.

The implant parses this file to locate the start and end addresses of the **libphp** mapping. These addresses are then used to constrain subsequent modifications to the intended target module. Instead of patching memory blindly, the implant identifies the executable **libphp** mapping and limits its changes to that region. This is achieved by following the steps:

```
read /proc/self/maps -> find libphp -> mprotect RWX -> patch relocations -> mprotect RX
```

Reading **/proc/self/maps** is not inherently malicious. Debuggers, profilers, and memory-management tools may legitimately inspect process mappings. However, it becomes far more suspicious when immediately followed by changes to memory permissions, relocation patching, and writes into executable regions of a targeted shared library.

This sequence is particularly anomalous in Apache worker processes, where workers have little reason to inspect their own memory mappings and then immediately modify executable pages.

Defender tip: Access to **/proc/self/maps** followed by **mprotect()** activity, relocation patching, or writes into executable memory regions are high-signal behaviors worth investigating, especially in Apache worker processes.

Redirecting execution inside libphp

Many defenders will be familiar with malware hooking via **LD_PRELOAD** or tampering with **PLT/GOT** entries. Our sample takes a more surgical approach, combining relocation-based hooking with direct rewriting of call targets inside **libphp**, so that selected operations are detoured into implant-controlled code first.

Before applying these modifications, the implant temporarily changes memory protections on the targeted **libphp** mapping, patches selected relocation and call targets, and then restores the original protections.

From an implementation perspective, the sample walks relocation data associated with **libphp** and adjusts PC-relative call targets for a small set of functions. From a defender's perspective, the precise relocation mechanics matter less than the outcome.

Within the PHP module context, the sample silently redirects calls that would normally invoke file and memory APIs like **open**, **close**, **mmap**, and **__fxstat**, and therefore gains control over how PHP opens, sizes, maps, and ultimately executes targeted script files. These hooks form the foundation for the in-memory web shell delivery mechanism described next. This also gives the threat actor another advantage: they don't need to inject a new shared library into the normal loader path.

Delivering a web shell through memory mapping

In our opinion, this was the most distinctive aspect of the sample. The implant intercepts attempts by PHP to open three specific script files:

- **apm_css.php3**
- **full_wt.php3**
- **webtop_popup_css.php3**

These filenames were likely chosen due to their existence within the targeted BIG-IP APM webtop environment, and are therefore unlikely to trigger suspicion.

When PHP opens one of these files, the implant records the file descriptor. When that file is subsequently memory mapped, the implant creates a modified in-memory view containing both the embedded web shell and the original script content.

The on-disk file does not need to contain the final web shell content at all; execution follows from the modified in-memory representation created by the implant at runtime.

```

v6 = a4;
v7 = a6;
if ( a5 <= 2 )
    return real_mmap(a1, a2, a3, v6, a5, v7);
if ( apm_css_fd == a5 )
{
    v8 = 0;
}
else if ( full_wt_fd == a5 )
{
    v8 = 1;
}
else
{
    v8 = 2;
    if ( webtop_popup_css_fd != a5 )
        return real_mmap(a1, a2, a3, v6, a5, v7);
}
v12 = &target_php_records[19 * v8];
if ( php_webshell_len + v12[2] > a2 )
    return real_mmap(a1, a2, a3, v6, a5, v7);
v10 = real_mmap(a1, a2, a3 | 2, 2, a5, a6);
v7 = a6;
v6 = a4;
if ( v10 == -1 )
    return real_mmap(a1, a2, a3, v6, a5, v7);
v11 = v10;
// Prepend the embedded webshell to the mapped PHP script,
// preserving the original script content after the payload.
move_original_php_forward(php_webshell_len + v10, v10, v12[2]);
copy_webshell_payload(v11, &php_webshell_payload, php_webshell_len);
return v11;
}

```

SOPHOS | X-OPS

Figure 5: The **mmap()** hook checks whether the mapping belongs to a tracked PHP script file descriptor. If so, it calls the original **mmap()** and prepends the embedded PHP web shell to the mapped content in memory, leaving the on-disk file unchanged.

The embedded PHP payload behaves like a classic web shell, with a few notable characteristics:

- Reads raw request bytes from **php://input**
- Checks for a short magic prefix (BSOHAzPB) at the start of the request body
- Decrypts the remainder using a small stream cipher
- Executes the decrypted content via **eval**

- Returns **HTTP status 201** and sets **Content-Type: text/css; charset=utf-8** to blend into normal asset requests

The payload is templated, with key strings rewritten at runtime – further complicating static signature matching. In our sample, the request marker (BSOHAzPB) and web shell key (wSLjN1beuR) were patched into the PHP at runtime rather than stored directly in their final form.

Defender tip: Web shell detection mechanisms should include runtime behavior and memory inspection, not just file scanning. For this class of threat, it is entirely possible for the on-disk PHP file to appear benign, while the in-memory mapping contains malicious code.

Delayed activation via `apr_time_now`

Rather than spawn threads or heavy routines during early startup, the implant uses its hook on **`apr_time_now`** as a delayed trigger.

When the Apache process begins making routine time calls, the implant spawns and detaches the worker thread responsible for creating the local UNIX socket backdoor. This timing minimizes the risk of destabilizing the service and helps the implant blend into normal runtime behavior. It also complicates dynamic analysis environments that only observe a brief window after process start.

As well as the HTTP-driven web shell, the implant also establishes a local AF_UNIX socket at **`/run/bigtlog.pipe`** rather than exposing a traditional TCP listener.

On 32-bit Linux, socket operations are commonly multiplexed through the historical **`socketcall`** system call, which handles operations such as **`socket`**, **`bind`**, **`listen`**, and **`accept`**. The implant uses this interface consistent with x86-32 environments.

After a short authentication check based on the token Kzwd6jM5, the implant redirects standard input, output, and error streams to the socket and executes **`/bin/bash`**, which provides interactive shell access without opening a TCP listening port, making it harder to detect using network-based monitoring alone. The listener remains active after successful authentication, using **`fork()`** to create a separate shell process while continuing to accept additional connections.

Interestingly, we couldn't locate any built-in mechanism that would allow the threat actor to connect to the socket. There was no client-side socket code, and no additional references to the authentication token, suggesting that the two methods (the web shell and the socket) are

separate capabilities. It's possible that the threat actor may be able to interact with the socket via the web shell, but we have no evidence to confirm or refute that at this time.

```
close();
dup2(v1, 0);
dup2(v1, 1);
dup2(v1, 2);
if ( v1 > 2 )
    close();
memcpy(v8, &enc_bin_bash, sizeof(v8));
rc4_decrypt_string(v8, 10);           // /bin/bash
v7[0] = &v8[5];
v7[1] = 0;
sys_execve(v8, v7, 0);
sys_exit();
v0 = -1;
```



Figure 6: The implant hands a local socket connection directly to **/bin/bash** by redirecting standard streams and executing the shell binary.

Defender tip: Local IPC endpoints, especially under **/run**, merit scrutiny in server-side compromises.

Protection and defense

Sophos detects this threat as **Linux/Agnt-IC**.

Threat hunting

These behaviors should be treated as investigatory leads and correlated with file-integrity, process, memory, and BIG-IP-specific evidence rather than used as standalone confirmation.

Web-layer signals

Check for:

- Requests to the **.php3** endpoints listed above, especially if they are rare in your environment.
- PHP endpoints returning **HTTP 201** while claiming to be CSS (**Content-Type: text/css; charset=utf-8**).
- Repeated POST requests with consistent structure or unusual body sizes to CSS-like PHP paths.

Host-level signals

Check for:

- Apache worker processes reading **/proc/self/maps**.
- Temporary changes to memory permissions on PHP module mappings (RWX followed by RX) around **libphp**.
- Creation of a UNIX domain socket at **/run/bigtlog.pipe**.
- Apache lineage processes redirecting **stdio** and executing **/bin/bash**.

Guidance for defenders

Incident response guidance

If this implant is suspected:

1. Capture volatile evidence first:
2. Assume dual access:
3. Service restart alone is not a guarantee:

Hardening and detection

While every environment is different, several broad measures can reduce exposure or improve visibility:

Attack surface reduction

- If .php3 execution is not required in your environment, consider disabling it after change-control and application-impact review. Do not apply this as a blanket mitigation to BIG-IP APM systems without following F5 guidance
- Prioritize configurations that minimize long-lived PHP execution paths where practical
- Restrict **ptrace** (this can reduce some cross-process inspection and injection opportunities, but is not necessarily a mitigation for in-process loader and patching behavior):

```
echo 1 > /proc/sys/kernel/yama/ptrace_scope
```

Add to Apache config:

```
<FilesMatch "\.php3$">  
    Require all denied  
</FilesMatch>
```

Minimize unnecessary PHP execution functionality and review the operational impact of restricting dangerous functions where appropriate.

Behavioral monitoring

- Alert on unusual combinations of activity, such as Apache workers:
 - Reading **/proc/self/maps**
 - Changing memory protections on **libphp**
 - Binding UNIX sockets under **/run**
 - Spawning **/bin/bash**
- Treat **HTTP 201 + text/css** responses from PHP endpoints as suspicious when they are not part of normal application behavior.
- **Plan for memory-aware response:**
 - Incorporate process memory collection and comparison of on-disk vs in-memory module contents into incident response playbooks for critical web servers.

Conclusion

This implant demonstrates how modern Linux malware can deliver familiar attacker capabilities through sophisticated delivery mechanisms. While the embedded PHP ultimately behaves like a traditional web shell, the surrounding infrastructure is considerably more advanced: custom ELF loading, early startup interception, APR-aware module monitoring, relocation patching, and memory-only payload delivery.

Based on our analysis of the related **umount** and infected **httpd** samples, we assess that this campaign involves a staged architecture. An installer component appears responsible for deployment, persistence, and propagation, while the infected httpd component focuses on runtime capability, process manipulation, web shell delivery, and interactive access.

Perhaps the most significant finding is that the web shell does not need to exist in its final form on disk. Instead, the implant alters how targeted PHP files are presented to the running process, meaning the content observed by Apache and PHP can differ from the content visible to traditional file-based inspection. As a result, responders who focus exclusively on the filesystem may overlook critical evidence.

This sample is part of a broader class of Linux threats that rely less on obvious on-disk artifacts and more on runtime manipulation. In such cases, defenders should focus on correlation across layers. Filesystem inspection alone may not reveal the compromise, but network telemetry, protocol inspection, process monitoring, memory analysis, and integrity verification can each expose different parts of the attack chain.