Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report NATS Messaging System 11.2018

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, BSc. J. Hector, J. Larsson, Dipl.-Ing. A. Inführ, MSc. D. Weißer, Dr. J. Magazinius

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"NATS Server is a simple, high performance open source messaging system for cloud native applications, IoT messaging, and microservices architectures."

From https://nats.io/

This report documents the results of a large-scale security assessment targeting the NATS compound. Carried out by Cure53 in November 2018 and funded by The Linux Foundation/ CNCF, this project revealed nine security-relevant findings on the NATS project's scope.

It needs to be underlined that the Cure53 team was tasked with performing both a wide-scoping penetration test and an in-depth source code audit of the NATS server software. In addition, the testing team assessed several, specially preselected NATS clients. In terms of resources, eight members of the Cure53 team contributed to these investigations and worked against a time budget of eigtheen days. As already noted above, the project was commissioned and sponsored by The Linux Foundation/ CNCF and it belongs to a longer series of assessments carried out by Cure53 through this particular scheme.

Zooming in on methods, Cure53 chose a two-pronged approach to testing against NATS. In order to maximize coverage, the testers performed a source code audit, as well as engaged in classic penetration testing against a NATS-provided cloud instance. After some back and forth, the maintainers of the NATS compound made some machines available to Cure53, so that efficient examinations could be executed. All in all, a holistic methodology made it possible for good coverage to be reached on both the server and the most relevant clients. It needs to be noted, however, that given the time constraints imposed for this project, not all clients could be audited. A selection was based on evaluating which clients are the most relevant from a security standpoint and subjecting those items to targeted examinations.

Over the course of the project, all communications were done on Slack. The NATS team invited Cure53 over to their instance and provided a dedicated channel for asking and answering any emerging questions about the scope. Key discoveries were discussed and Cure53 reported on the progress of the test in real-time.

Among the aforementioned nine issues found on the scope, five were classified to be security vulnerabilities and four were noted as constituting general weaknesses with considerably lower exploitation potential. On the other end of the severity spectrum, one should note one issue ascribed with a "*Critical"* risk potential. This is due to the fact that

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

the problem would empower an attacker to the point of granting him/her a capacity to execute arbitrary code on the targeted victim's system.

In the following sections, the report first describes the scope and then elaborates on the test methodology in order to clarify which tasks and steps have been completed during this assessment. Next, all spotted issues are documented, with the tickets supplying technically-focused PoCs for ease of verification and further in-house work. What is more, mitigation and fix advice for moving forward is also included for each discovery. In the final section, drawing on the findings of this November 2018 project, Cure53 offers some broader conclusions and notes on the impressions gained about the state of security and the perceived maturity-level reached by the examined NATS compound.

Scope

- **NATS Open Source Messaging System**
  - Cure53 made use of the sources publicly available online on GitHub (*nats-io*)
  - The chosen branches that the Cure53 tested against were
    - https://github.com/nats-io/gnatsd `branch master commit 0d13c90...`
    - https://github.com/nats-io/nkey `branch master commit f9a6cff...`
    - https://github.com/nats-io/jwt `branch master commit 98f5dca...`
  - Cure53 was further supplied with additional information via a private, instructive GitHub repository.
  - Cure53 further investigated against several of the NATS clients in scope and conducted penetration tests and source code audits (i.e. for *go-nats*, node-*nats, java-nats* and *cnats*):
    - https://github.com/nats-io/go-nats `branch master commit 6379777...`
    - https://github.com/nats-io/node-nats `branch master commit c1ed196...`
    - https://github.com/nats-io/java-nats `branch master commit f49af69...`
    - https://github.com/nats-io/cnats `branch master commit ea0e97b...`

Test Methodology

This section describes the methodology that was used during this source code audit and penetration tests. The test was divided into two phases with corresponding two-fold goals and focal points that were closely linked to the areas referenced in the scope. The first phase (Phase 1) mostly entailed manual source code reviews. The review carried out during Phase 1 aimed at spotting insecure code constructs. These were marked for potential capacity of leading to memory corruption, information leakages and other similar flaws. The second phase (Phase 2) of the test was dedicated to classic penetration testing. At this stage, it was examined whether the security promises made by NATS in fact hold against real-life attack situations and malicious adversaries.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Phase 1 (Manual Code Auditing)

A list of items below seeks to detail some of the noteworthy steps undertaken during Phase 1, which entailed the manual code audit against the sources of the NATS software in scope. This is to underline that, in spite of the relatively low number of findings, substantial thoroughness was achieved and considerable efforts have gone into this test. The tasks completed during the project are listed next.

- The security of the nonce generation randomness of *gnatsd* was pondered and the nonce signing for *nkeys* usage was audited. The potential issue, entailing the sole nonce signing instead of relying on all *CONNECT* message fields, was eventually dismissed.
- The *account registration* aspect of the server was audited, especially with respect to options for potentially re-registering an account, yet this proved invulnerable.
- The server-side permission checks for publishing and subscribing, as connected to the matching of subjects, was reviewed but was found safe.
- The construction of permission sets and subject lists within *gnatsd* was analyzed but did not expose any weaknesses.
- The *auth_required* mechanism of the server was unsuccessfully investigated for being somehow bypassable.
- The protocol parsing, along with the privilege model of *gnatsd*. were checked for conceptual problems but none could be spotted.
- The logger functions within the server were audited for sinks that could potentially disclose unwanted security-relevant information in clear-text. It was found that the logger functions have special filters to prevent this type of information leaks.
- *gnatsd* was investigated for command execution, namely taking user-controllable input to invoke a shell or other stacking of commands. This did not yield.
- General input sources were checked for being vulnerable in respect to initial attack vectors or facilitating further attacks.
- The *jwt* token parsing was checked for flaws, including classical attack algorithm stripping and password brute-forcing.
- The data structures of connected clients were audited for potential overflows. Although it is theoretically possible to achieve integer overflows in certain situations, the abuse of such cases is not feasible in any probable scenarios.
- The hand-rolled parser logic looked promising in terms of compromise at first. Even though it was difficult to comprehend, it proved to be impenetrable.
- The publishing and subscription mechanisms were audited as regards supported wildcards. The defined permissions worked as expected despite the application of the discussed patterns.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- The separation of client and route connections within the server was thoroughly investigated and no problems could be found in this realm.
- The security of the *node.js*-client's *node-nats* was assessed. Besides potential for a Denial-of-Service via a malicious JSON structure, the client is quite small and did not offer much of an attack surface.
- The cryptographic primitives used by *nkeys* were investigated. As *nkeys* basically wraps the external Go implementation of the *ed25519* signing algorithm, no issues could be determined in the implementation.
- Furthermore, the cryptographic functions within *jwt* were audited. Since *jwt* uses the *nkeys* for signing tokens, no issues with the implementation were found.
- The java-client called *java-nats* is rather small and therefore did not offer too much of an attack surface. A brief assessment of this client did not uncover any issues.
- The java *nkeys* implementation is based on the respective Go version. An in-depth analysis of the cryptographic primitives was unfortunately not possible within the given timeframe. Drawing on code comparison. it should be as secure as the original one.
- The protocol parsing and successive usage of the the received information within the *cnats* library was audited and led to the filing of NAT-01-003.
- The JSON parser of the c-client was audited for problems and eventually uncovered issue NAT-01-004.
- The JSON parser contained in *util.c* of *cnats* was audited for typical problems but did not lead to any exploitable scenarios.
- The *c-client* protocol *parser.c* was audited for weaknesses but proved to be strong and durable.
- The hardening flags of *cnats* were investigated for proper configuration of the respective system mechanisms.

Phase 2 (Code-Assisted Penetration Testing)

A list of items below seeks to detail some of the noteworthy steps undertaken during the second phase of the test, which encompassed code-assisted penetration testing against the NATS system in scope. Given that the manual source code audit did not yield an overly large number of findings, the second approach was added as means to maximize the test coverage. As for specific tasks taken on to enrich this Phase, these can be found listed and discussed in the ensuing list.

- The *go-nats* client code was modified to facilitate sending of custom messages, effectively aiming to examine server-side handling of such content.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- Using the modified *go-nats*, the respective *gnatsd* code was checked for proper handling of unexpected characters. As an eventuality, *logfile* poisoning was studied.
- The *go-client* was investigated for compliance with server requirements (authentication, TLS initiation, etc.) but the server was found to simply and effectively terminate non-compliant connections after a timeout.
- The *cnats* client library was thoroughly examined with the help of modified implementations made on the provided examples. These were combined with running a customized fuzzer.
- Several fuzzed heap overflows were investigated on the basis of the created *ASAN* report. This was meant to gauge feasibility of exploiting their respective root causes.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *NAT-01-001*) for the purpose of facilitating any future follow-up correspondence.

NAT-01-001 *gnatsd:* Tracing allows *logfile* injection *(Low)*

A minor issue was discovered in the way that *gnatsd* handles the process of printing to *log files.* For example, a client that publishes a message can include newlines in its payload and these newlines will be left unsanitized during trace-logging. This way, it is possible to poison the *logfile*, hide malicious exploitation attempts, or otherwise confuse anyone studying the logs. The vulnerable sink was originally found in the following part of the application's source.

Affected File:
gnatsd/server/client.go

Affected Code:

```
[...]
func (c *client) traceMsg(msg []byte) {
	if !c.trace {
		return
	}
	// FIXME(dlc), allow limits to printable payload
	c.Tracef("<<- MSG_PAYLOAD: [%s]", string(msg[:len(msg)-LEN_CR_LF]))
}
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
[...]
```

As one can see above, *%s* is used to directly display *msg* without any prior sanitization, so that newlines will still be carried over to the *logfile*. This can be reproduced with the *golang* statement supplied next.

PoC Code:

```
[...]
nc.Publish(subj, []byte("HELLO FROM CURE]\n[#1337] Received on [INJECTED]:
Malicious["))
[...]
```

Sending a message provided here will result in the creation of particular content in the *logfile.* This can be consulted next.

Shell Excerpt:

```
$ tail /home/nats/log/gnatsd.log
[15818] 2018/11/12 11:05:01.307636 [TRC] 165.227.157.116:46098 - cid:13175 - <<-
MSG_PAYLOAD: [HELLO FROM CURE]
[#1337] Received on [INJECTED]: Malicious[]
[15818] 2018/11/12 11:05:01.307650 [TRC] 165.227.157.116:46098 - cid:13175 - <<-
[PING]
```

It is recommended to always convert newlines before printing them into *logfiles*. Note that this issue might extend beyond trace-logging of messages and can potentially affect any function that dangerously prints user-controlled messages.

NAT-01-004 cnats: DoS and memory leakage due to type-confusion *(Medium)*

After the *INFO* JSON is received, it is then parsed and later accessed to extract the necessary information. When accessing a value, a type confusion can be triggered. This, in turn, allows a malicious server to specify an arbitrary address used to read memory, therefore either leaking memory or crashing the application due to unmapped memory.

A field that contains a number as value is internally represented as *TYPE_NUM*. However, if the field is expected to have a string, as is for instance the case with the '*version'* field in the *INFO* JSON, a *TYPE_STR* is requested. If a *string* value is requested yet the field contains a number, the check that ensures that the requested type matches the field type does not catch the flaw. This is due to the last part of the *if-statement* highlighted below.

Affected File:
cnats/src/util.c

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected Code:

```
[...]
natsStatus
nats_JSONGetValue(nats_JSON *json, const char *fieldName, int fieldType, void
**addr)
{
[...]
    // Check parsed type matches what is being asked.
    if ((((fieldType == TYPE_INT) || (fieldType == TYPE_LONG)) && (field->typ !=
TYPE_NUM))
        || ((field->typ != TYPE_NUM) && (fieldType != field->typ)))
    {
        return nats_setError(NATS_INVALID_ARG,
                             "Asked for field '%s' as type %d, but got type %d
when parsing",
                             field->name, fieldType, field->typ);
    }
[...]
```

Although the requested type does not match the field type, the *and-condition* that the field type is not *TYPE_NUM* breaks this statement. The result of this operation is presented next.

Evaluated IF part:

```
[...]
(TYPE_NUM != TYPE_NUM) && (TYPE_STR != TYPE_NUM)
[...]
```

The statement furnished above will never evaluate to *true,* thus the mismatch is not caught. A Proof-of-Concept (PoC) below demonstrates the problem.

PoC:

```
% echo -e 'INFO
{"server_id":"NATP5XGXVZQXNH2ZA6OOLGCTAVOLD6DP7BAHRSOGEKRIMS22TDVOFP6U","version
":16045690984833335023}\r\n' | nc -lvp 1243

% ./nats-subscriber -s  nats://127.0.0.1:1243
Listening asynchronously on 'foo'.
zsh: segmentation fault (core dumped)  ./nats-subscriber -s
nats://127.0.0.1:1243
```

The application crashes due to *strdup* trying to read from *0xdeadbeefdeadbeef* (i.e. a hexadecimal representation of the *integer* value in the *version* field).

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Having multiple types that can be requested (*TYPE_INT*, *TYPE_LONG*, etc.) for numbers, yet then using only one number-type during parsing (*TYPE_NUM*) makes this check a little more complicated. It is recommended to either rewrite the parsing to use congruent types or to rework the *if-condition.*

NAT-01-005 cnats: Stack buffer overflow can lead to RCE *(**Critical**)*

Another issue discovered on the scope could lead to a stack corruption as a result of network-provided data being handled incorrectly. This can translate to a Denial-of-Service issue but can also be leveraged to signify Remote Command Execution (RCE). The issue resides in the protocol handler for messages representing the "*MSG"* type and can be triggered by providing too many arguments. The procedure is shown in the following code snippet.

Affected file:
cnats/src/parser.c

Affected code:

```
[...]
static natsStatus
_processMsgArgs(natsConnection *nc, char *buf, int bufLen)
{
[...]
    int             index   = 0;
[...]
    struct slice    slices[4];                  \\ [1]

    for (i = 0; i < bufLen; i++)
    {
        b = buf[i];

        if (((b == ' ') || (b == '\t') || (b == '\r') || (b == '\n'))
            && started)
        {
            slices[index].start = start;            \\ [2]
            slices[index].len   = len;
            index++;
[...]
    }
    if (started)
    {
        slices[index].start = start;            \\ [3]
        slices[index].len   = len;
        index++;
    }
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The function reserves an array of the *slice* type, which can hold four elements on the stack in order to store pointers and lengths of the arguments [1]. However, the index is not limited when the provided data is being parsed, thus allowing a server to send a message with more than four arguments. This effectively causes a stack-based buffer overflow [2][3].

It is possible to verify the vulnerability with the use of the *nats-publisher* test application. First, a very simple server needs to be started with *netcat* and it should send packets to a client with a short delay. Secondly, the *nats-publisher* example application can be executed in order to trigger the issue. Please note that the bug resides in the library and not in the demo application itself.

PoC:

```
(sleep 2; echo -e 'INFO
{"server_id":"NDWUNZMDM.0.0","port":4222,"max_payload":1048576,"cl{ent_Jd":11035
7}\r' ; sleep 1; echo -e "PONG\r" ; sleep 1; perl -e 'print "MSG " . "a "x5 . "\
r\n"') | nc -lvp 1241

nats-publisher -s nats://127.0.0.1:1241 -count 10
*** stack smashing detected ***: <unknown> terminated
zsh: abort (core dumped)  ./nats-publisher -s nats://127.0.0.1:1241 -count 10
```

If the application is compiled with a modern compiler, a stack corruption is detected and the process is killed, therefore making it harder to exploit this vulnerability. However, when an older compiler or a 32-bit architecture is in use, this issue lets an attacker overwrite the program's instruction pointer easily. The latter effectively means that RCE can be accomplished.

It is recommended to limit the number of elements that can be written into the *slices* structure to the actual size of the available array. In order to make the exploitation of this kind of problems harder in general, it is further advised to apply the fix described in NAT-01-006.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NAT-01-006 cnats: Lack of compiler hardening flags *(Low)*

Using tools like *checksec*[1] or *PEDA*'s[2] built-in functionality to check for basic hardening support reveals that the default compiler options omit *PIE*[3], full *RELRO*[4] and stack canaries when building *cnats* from a source with older versions of *GCC*. This can be observed next.

```
$ gdb libnats.so.1.8
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : Dynamic Shared Object
RELRO     : disabled
```

While modern compilers enable the advised flags by default, older compilers, for instance the default *GCC* version installed on Debian 8, calls for the security features to be enabled explicitly.

First up, *PIE* renders the exploitation of memory corruption vulnerabilities a lot more difficult. This can be attributed to the fact that additional information leaks are required to conduct a successful attack. Secondly, *RELRO* marks different binary sections - like the *GOT* - as read-only. Therefore, it kills a handful of techniques that come in handy when attackers somehow gained the capability to arbitrarily overwrite memory. This is why it is strongly recommended to add the discussed compiler flags to the Makefile in a manner presented next.

CFLAGS:

```
CFLAGS='-Wl,-z,relro,-z,now -pie -fPIE -fstack-protector-all -D_FORTIFY_SOURCE=2
-O1'
```

NAT-01-008 node-nats: DoSe via malicious JSON structure *(Low)*

The *NodeJS* client can define that any message payloads are converted from/to JSON. This is achieved by setting *json* parameter to *true* in the *connect* function. It was discovered that a possibility to trigger an exception nevertheless exists and can lead to a crash of the client. This is triggered by sending an object which overwrites the *toString* function with a string. As soon as the received message is used as a string in any context, an exception is triggered.

[1] http://www.trapkit.de/tools/checksec.html
[2] https://github.com/longld/peda
[3] https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html
[4] http://tk-blog.blogspot.de/2009/02/relro-not-so-well-known-memory.html

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Malicious client:

```
[...]
var NATS = require('nats');
var nats = NATS.connect({'url':"nats://nats-audit.synadia.io:4222",
'user':'test', 'pass':'test', "tls":true});

// Simple Publisher
nats.publish('foo','{"toString": ""}');
[...]
```

Victim-listening:

```
[...]
var NATS = require('nats');
var nats = NATS.connect({'url':"nats://nats-audit.synadia.io:4222",
'user':'test', 'pass':'test', "tls":true,"json": true});

// Simple Subscriber
nats.subscribe('foo', function(msg) {
console.log("There is a message for you")
console.dir(msg)
console.log(msg+"");  // Crash happens here
});
[...]
```

Output for Victim:

```
There is a message for you
{ toString: '' }
TypeError: Cannot convert object to primitive value
```

It could be taken into consideration to use *hasOwnProperty*[5] to check if *toString* was overwritten by the received JSON object. In case *toString* has been tampered with, the message could be rejected to protect the client from experiencing a crash later on.

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript...Objects/Object/hasOwnProperty

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

NAT-01-002 gnatsd: Parsing issues due to newlines in *subscriber* messages *(Info)*

An issue directly connected to NAT-01-001 can be found in the relevant clients, e.g. in *nats-sub*. Although they do not represent literal clients that are used in production on every instance of *NATs*, it is possible to demonstrate that unsanitized printing of messages - as illustrated in NAT-01-001 - can have further consequences. For example, listening to *foo* as a subject and directly printing out the messages might cause some parsing issues when the message includes newlines again. In that scenario, it might confuse a client to receive messages on completely unrelated subjects, to which they should not even have access to.

PoC Code:

```
[...]
nc.Publish(subj, []byte("HELLO FROM CURE'\n[#1337] Received on [INJECTED]:
'Malicious"))
[...]
```

Shell Excerpt:

```
$ nats-sub -s test:test@nats-audit.synadia.io "foo"
Listening on [>]
[#1] Received on [foo]: 'HELLO FROM CURE'
[#1337] Received on [INJECTED]: 'Malicious'
```

Although the issue is of extremely limited severity because it only concerns the printing of the message and not the actual parsing of the subject line, it was still found important to highlight the consequences of allowing newlines to be left unsanitized. It is recommended to encode newline characters prior to having the received messages printed.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NAT-01-003 cnats: Missing NULL termination on Visual Studio below 2015 *(Low)*

Visual Studio versions before 2015 do not supply a *snprintf* function and *_snprintf* (with an underscore) is used instead. However, *_snprintf* does not add a terminating *null* byte should the input be equal or greater in size than the destination buffer. This behavior is documented in the official Microsoft documentation[6]:

> *If len = count, then len characters are stored in buffer, no null-terminator is appended, and len is returned.*
> *If len > count, then count characters are stored in buffer, no null-terminator is appended, and a negative value is returned.*

Since most string operations in C are based on a terminating *null* byte, this could lead to unexpected behaviors, such as *strlen* returning a value exceeding the actual buffer. The code excerpt below shows how a malicious server is able to send an *INFO* command, triggering the missing *null* termination. Please keep in mind that there is an underlying assumption here that the client has been compiled with a Visual Studio version earlier than 2015.

Affected File:
cnats/src/srvpool.c

Affected Code:

```
[...]
natsStatus
natsSrvPool_addNewURLs(natsSrvPool *pool, const natsUrl *curUrl, char **urls,
int urlCount, bool *added)
{
[...]
        while ((s == NATS_OK) && natsStrHashIter_Next(&iter, &curl, NULL))
        {
[...]
            sport = strrchr(curl, ':');
            portPos = (int) (sport - curl);
            if (((nats_strcasestr(curl, "localhost") == curl) && (portPos == 9))
                    || (strncmp(curl, "127.0.0.1", portPos) == 0)
                    || (strncmp(curl, "[::1]", portPos) == 0))
            {
                isLH = ((curl[0] == 'l') || (curl[0] == 'L'));

                snprintf(url, sizeof(url), "localhost%s", sport);
                found = (natsStrHash_Get(pool->urls, url) != NULL);
                if (!found)
                {
```

[6] https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/2ts7cx93(v=vs.110)

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
                    snprintf(url, sizeof(url), "127.0.0.1%s", sport);
                    found = (natsStrHash_Get(pool->urls, url) != NULL);
                }
[...]
```

By sending a *connect_urls* entry and replacing the port of the URL part, one is left with a string of a *length* that exceeds the *url* buffer. The *snprintf* call will leave the *null* termination of the *url* buffer out.

The relevant part of the *INFO* command is furnished next.

***INFO* JSON from malicious server:**

```
INFO {[...]"connect_urls":["localhost:<string that exceeds 256 bytes>"][...]}
```

It was not possible to exploit this flaw in any meaningful way during this assessment, hence its placement in the "*Miscellaneous"* section. However, it is still recommended to ensure that *null* termination of strings follows good coding practices.

NAT-01-007 cnats: Uncleared memory leads to OOB memory read *(Low)*

An issue was discovered that can lead to an OOB memory read on the heap. Since it was not proven that the bug is exploitable, this issue was categorized as *Misc* and *"Low"*. A PoC was created and resulted in an ASAN error report supplied below.

ASAN:

```
% ./publisher -s  nats://127.0.0.1:1241 -count 10
Sending 10 messages to subject 'foo'
Error: 26 - Timeout
Error: 26 - Timeout - (conn.c:2978): Timeout
Stack: (library version: 1.8.0)
  01 - _flushTimeout
  02 - natsConnection_FlushTimeout
=================================================================
==7585==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61b000000659
at pc 0x5626c6616fcc bp 0x7f4d7f5f2b30 sp 0x7f4d7f5f22e0
READ of size 1498 at 0x61b000000659 thread T4
    #0 0x5626c6616fcb in printf_common(void*, char const*, __va_list_tag*)
(/tmp/na/asan/build/examples/publisher+0x62fcb)
    #1 0x5626c661845d in __interceptor_vsnprintf
(/tmp/na/asan/build/examples/publisher+0x6445d)
    #2 0x5626c6724ad6 in nats_setErrorReal /tmp/na/asan/src/nats.c:1362:5
    #3 0x5626c67458ba in nats_JSONParse /tmp/na/asan/src/util.c:709:26
    #4 0x5626c6703522 in _processInfo /tmp/na/asan/src/conn.c:509:9
    #5 0x5626c67031ed in natsConn_processAsyncINFO
/tmp/na/asan/src/conn.c:588:12
    #6 0x5626c6732edd in natsParser_Parse /tmp/na/asan/src/parser.c:734:25
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
    #7 0x5626c6718380 in _readLoop /tmp/na/asan/src/conn.c:1844:17
    #8 0x5626c674aca9 in _threadStart /tmp/na/asan/src/unix/thread.c:39:5
    #9 0x7f4d846d4a9c in start_thread (/usr/lib/libpthread.so.0+0x7a9c)
    #10 0x7f4d842bab22 in __GI___clone (/usr/lib/libc.so.6+0xfbb22)

0x61b000000659 is located 0 bytes to the right of 1497-byte region
[0x61b000000080,0x61b000000659)
allocated by thread T4 here:
    #0 0x5626c66bec88 in realloc
(/tmp/na/asan/build/examples/publisher+0x10ac88)
    #1 0x5626c674c253 in natsBuf_Expand /tmp/na/asan/src/buf.c:136:19

Thread T4 created by T0 here:
    #0 0x5626c66129f2 in pthread_create
(/tmp/na/asan/build/examples/publisher+0x5e9f2)
    #1 0x5626c674ab25 in natsThread_Create /tmp/na/asan/src/unix/thread.c:67:15
```

The issue occurs because the allocated memory is not always cleared and the *NULL* terminations are missing. In the *natsParser_Parse()* function of *parser.c*, a state machine iterates over the received lines in order to parse messages and call corresponding functions. The following code snippet shows the handler for *INFO* protocol messages. While in *INFO_ARG* **[1]** state, the state machine iterates over the received string and appends each byte to a *nc->ps->argBuf* at **[5]** until a newline is found at **[2]**. Then, *nc->ps->argBuf* is assigned to *start* at **[3]** and *natsConn_processAsyncINFO()* **[4]** is called.

Affected file:
parser.c

Affected Code:

```
[...]
case INFO_ARG:                                                  \\ [1]
            {
                switch (b)
                {
[...]
                    case '\n':                                  \\ [2]
                    {
                        char    *start = NULL;
                        int     len    = 0;

                        if (nc->ps->argBuf != NULL)
                        {
                            start = natsBuf_Data(nc->ps->argBuf);   \\ [3]
                            len   = natsBuf_Len(nc->ps->argBuf);
                        }
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
                        natsConn_processAsyncINFO(nc, start, len);          \\ [4]
[...]
                        break;
                    }
                    default:
                    {
                        if (nc->ps->argBuf != NULL)
                            s = natsBuf_AppendByte(nc->ps->argBuf, b);      \\ [5]
                        break;
                    }
                }
[...]
```

Executing *natsConn_processAsyncINFO()* **[4]** eventually leads to a call being made to *nats_JSONParse()* and there the *buffer start* is passed as *jsonStr* to the function.

Affected file:
util.c

Affected Code:

```
[...]
natsStatus
nats_JSONParse(nats_JSON **newJSON, const char *jsonStr, int jsonLen)
{
[...]
  case JSON_STATE_NEXT_FIELD:
  {
  // We should have a ',' separator or be at the end of the string
  if ((*ptr != ',') && (*ptr != '}'))
  {
    s = nats_setError(NATS_ERR, "missing separator: '%s' (%s)", ptr, jsonStr);
[...]
```

If the provided JSON string is invalid, an error is displayed by the *nats_setError()* function. As illustrated, the strings including the arguments are passed to *nats_vsnprintf()* so that an error message can be built. Since *jsonStr* is not *null*-terminated, *vsnprintf()* will read memory outside of the allocated area of *jsonStr*.

Affected file:
nats.c

Affected code:

```
[...]
natsStatus
nats_setErrorReal(const char *fileName, const char *funcName, int line,
natsStatus errSts, const void *errTxtFmt, ...)
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
{
  char        tmp[256];
[...]
    va_start(ap, errTxtFmt);
    nats_vsnprintf(tmp, sizeof(tmp), errTxtFmt, ap);
    va_end(ap);
[...]
```

This behavior is caused by two different issues. First, the buffer passed to *natsConn_processAsyncINFO()* **[4]** is not *null*-terminated. Secondly, *natsBuf_AppendByte()* **[5]** does not clear allocated memory. For these reasons, it is recommended to *null*-terminate any buffer that is interpreted as string at any point. Moreover, flushing allocated memory is also advised.

NAT-01-009 java-nats: Internal version string does not reflect build version *(Info)*

It was found that the client version defined in the source code does not reflect the overall *jar* version defined in the build configuration. While this is not a problem *per se,* it can lead to confusion when one is checking for version strings in execution environments, especially while looking for incompatibilities during debugging.

Affected file:
java-nats/src/main/java/io/nats/client/Nats.java

Affected code:

```
[...]
public class Nats {

    /**
     * Current version of the library - {@value #CLIENT_VERSION}
     */
    public static final String CLIENT_VERSION = "2.1.1";
[...]
```

Affected file:
build.gradle

Affected code:

```
[...]
def versionMajor = 2
def versionMinor = 3
def versionPatch = 0
def versionModifier = ""
def jarVersion = "2.3.0"
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The flaw is caused by manually maintaining two version strings of the same inherent value. It is recommended to automatically fix up the version string in the client during the build process, therefore making sure that the respective version numbers are in sync.

Conclusions

The results of this Cure53 assessment of the NATS compound demonstrate that the tested scope is secure and robust. In other words, the outcomes of this November 2018 project commissioned by The Linux Foundation/ CNCF ascertain that NATS is being developed with a great deal of security awareness.

Despite considerable efforts aimed at finding a compromise with the use of a two-pronged approach, eight members of the Cure53 team only managed to spot and document nine security-relevant discoveries. What is more, neither the large-scale scoping nature, nor a significant budget of eighteen days dedicated to the project, managed to change this outcome and the number of security-problems in NATS needs to be evaluated as low.

To give some more notes on the context, it needs to be stated that the scope delineated for this test and the architecture of the application were well-defined and comprehensively communicated. Further, reasonably broad access to a dedicated test system was arguably supplied to the external, Cure53 testers. The assessment status and progress were reported on a dedicated messaging channel provided by NATS through Slack. All communications were fluent and quite productive. As an intermediate form of communication, a preview of the discovered issues was shared in advance of the final report's delivery.

After an initial sighting of the documentation and the provided test setup, Cure53 executed the project by auditing and penetration testing against the *gnatsd* application server, along with the primary *go-nats* client. What is more, an explicit focus was also placed on the recently added *nkeys* and *jwt* authentication components. During the later added Phase 2, the *node.js-client* called *node-nats*, the *java-client* called *java-nats* and the *c-client* called *cnats* were all reviewed, analyzed and tested. Finally, the cryptographic aspects of the *java-client* were only partially audited because of the pre-existing timing constraints.

Moving on to technical details, the *gnatsd* application server, as well as the primary *go-nats* client, should be judged as well-written. The strategically correct choice of the Go language made it extremely difficult to find any sort of memory corruptions or similar bugs during this assignment. At the end, the testers concluded that no such issues can

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

be documented. Similarly, the core code was also rather cleanly written, which further eased the process of the code review.

Cure53 found that the provided client library examples were minimal in their implementations, therefore exposing very little attack surface as far as code auditing and penetration testing was concerned. However, one "*Critical"*-ranking finding in the *C* implementation of the client, paired with a few less severe findings in this particular client, gives Cure53 some cause for concern about the state of security in this item.

As security issues in Go implementations are mostly logic flaws and occasionally race conditions, not much could be discovered within *gnatsd* and *go-nats*. This is because neither of them has complicated logic. Just to clarify, NATS is a relatively simple protocol for publishing and subscribing to messages via subjects. As clients connect and receive messages, these are in turn presented to the user. In this context of operations, the findings prove that the Go aspects of NATS comprise a robust platform that can be recommended for general deployment.

Besides resolving and addressing problems documented in this report, Cure53 strongly recommends that certain steps should be taken by NATS. Firstly, it is advised for the cryptographic implementation of *nkeys* within the *java-nats java-client* to undergo a dedicated audit. With the available resources, it was not possible to examine this component in great depth. Therefore, it is believed that some more attention should be given to this realm. Secondly, it is proposed that the developers continue looking for problems similar to the ones identified by Cure53. This would further improve the security of the platform and also means expanding on the scope of investigations and covering these aspects of the platform that could not be explored here. More specifically, the clients that were not subject to audits and testing should be revisited to ensure secure operations. Thirdly, it is recommended to extend the existing documentation within the source code of the application to facilitate future maintenance in practical terms.

To conclude, Cure53 positively evaluates the security posture found on the NATS scope during this November 2018 assessment funded by The Linux Foundation/ CNCF. The discoveries clearly show that that NATS generally offers a solid, recommendation-worthy solution. The small caveat should be applied to the *C* implementation, which definitely calls for some additional care and security-investment before being in a state that is safe enough for public consumption. In comparison, the NATS items pertaining to *Go* clearly meet or exceed goals and thresholds typically set for a project that can be considered secure.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Cure53 would like to thank Colin Sullivan, Derek and Ginger Collison as well as Stephen Asbury from the NATS team. Cure53 would further like to thank Chris Aniszczyk of The Linux Foundation, for the excellent project coordination, support and assistance, both before and during this assignment. Special thanks and gratitude also need to be extended to The Linux Foundation for sponsoring this project.