Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Pentest-Report Chrony 08.2017

Cure53, Dr.-Ing. M. Heiderich, BSc. D. Weißer, MSc. N. Krein, M. Wege

Index

Introduction

“Chrony is a versatile implementation of the Network Time Protocol (NTP). It can synchronize the system clock with NTP servers, reference clocks (e.g. GPS receiver), and manual input using wristwatch and keyboard. It can also operate as an NTPv4 (RFC 5905) server and peer to provide a time service to other computers in the network.”

From https://chrony.tuxfamily.org/

This document describes a combined manual code audit and a partially automated penetration-test of the Chrony NTP client and server. The project was undertaken by three testers from the Cure53 team over a timespan of eleven days in August of 2017.

The assessment’s focus was on the general software setup and several of the software interfaces, including system and network. In addition, the Cure53 testers had a close look at the core input and output implementation. A more detailed description of the scope can be found in a later section of this report.

The tests mostly attempted to locate vulnerabilities that would allow a malicious attacker to take over a client or a server by using logical flaws in the implementation, problems in memory handling, and bypassing authentication or control mechanisms. The software

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

version used for testing was the master branch (signified by the commit detailed below) of the main repository, dated on the project's first day.

Despite considerable effort by the Cure53 team, no major vulnerabilities were discovered. It goes without saying that the testers are impressed by the quality of the software. Only two minor issues detailed below were identified.

Scope

- **Chrony 3.2 (master)**
  - git://git.tuxfamily.org/gitroot/chrony/chrony.git
  - (commit 554b9b06de3cc17187cac2f5b0b7d7fc40d161c2)
- **Detailed Scope Information**
  - Build environment.
  - System level interface.
  - Core input/output functions.
  - Authentication mechanisms.
  - Client/server command interfaces.
  - Client NTP protocol implementation.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Test Methodology

The following two sections describe the approaches employed to finding vulnerabilities in the software package during the two individual phases. The first section describes which aspects of the code where covered during the manual code audit, while the second elaborates upon the approach taken for the code-assisted penetration test.

Part 1 (Manual Code Auditing)

The items listed below detail the activities performed during the first part of the assignment. It has to be mentioned that even though the manual code audit did not reveal the expected number of findings, significant efforts, creative ideas and time resources were dedicated into this particular aspect of the project.

- The existing documentation was examined and compared with the already existing knowledge about the problem subject's domain within the team.
- The binaries resulting from a build were checked for appropriate hardening features. Together with the rare application of *seccomp,* the baseline security was found to be exemplary.
- Manual style taint-analysis was applied to locate memory corruptions due to insufficient handling of potentially malicious data.
- Command handlers and core input/output were audited in an attempt to unearth improper sanitization of data.
- General memory, as well as array handling in particular, were checked for common mistakes. The realm revealed up to par standards and implementations.
- System-level interfaces and wrappers were investigated for erroneous usage and unhandled edge cases.

Part 2 (Code-Assisted Penetration Testing)

The following list presents more detail on the individual steps taken during the second part of the assessment. When compared to other recent security assessments of similar software packages, the manual code auditing phase did not yield the expected number of vulnerabilities. Therefore, additional measures were taken in the hope of uncovering weaknesses left undiscovered by the initial approach.

- The source code was used as a reference to identify potential problems in a white-box pentesting approach.
- Both client and server were built and deployed on several platforms so the tests could be run against their respective interfaces.
- Simple *printf()*-style logging and process inspection was used to single out interesting and frequently used code fragments.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

- It was attempted to crash the software via fuzzing. This was done by patching the server code to read client packets, process them, and exit.
- A secondary approach to fuzzing was implemented by preening the system interfaces at link-time and getting the software to use standard input/output.
- A sizable body of unauthenticated client packets was captured and mutation-based fuzzing via AFL was utilized with the goal being to crash the server during handling.

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. Each finding is given a unique identifier (e.g. *CHR-01-001*) for the purpose of facilitating any future follow-up correspondence.

CHR-01-001 chronyc: Null Pointer Deref in *manual list* Response Handler *(Low)*

An issue was discovered that allows a malicious Chrony server to trigger a null-pointer dereference in the Chrony client. This is accomplished by sending an invalid response to the "*manual list*" command which is handled by the *process_cmd_manual_list* function. The responsible code is furnished in the code snippets below. In this example, a loop iterates over the entries provided by the server.

Affected File:
/chrony/client.c

Affected Code:

```
process_cmd_manual_list(const char *line)
{
[...]

  for (i = 0; i < n_samples; i++) {
    sample = &reply.data.manual_list.samples[i];
    UTI_TimespecNetworkToHost(&sample->when, &when);

    print_report("%2d %s %10.2f %10.2f %10.2f\n",
                 i, UTI_TimeToLogForm(when.tv_sec),
```

The contents of the *reply* variable are controlled by the server, meaning that they allow to fully control *sample*. Therefore it is also possible to control *when.tv_sec,* which is passed to the *gmtime* function.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Affected File:
/chrony/util.c

Affected Code:
```
char *
UTI_TimeToLogForm(time_t t)
{
  struct tm stm;
  char *result;

  result = NEXT_BUFFER;

  stm = *gmtime(&t);
  strftime(result, BUFFER_LENGTH, "%Y-%m-%d %H:%M:%S", &stm);
```

If an invalid value is passed to *gmtime,* a *NULL* pointer is returned. This leads to a crash due to the missing return value check.

PoC (traffic from MiTM proxy):
```
Client to server:
06010000002900000f788ed690000....
Server to client:
06020000002900000000000000000000f788ed690000000000000000000000000161616161616161
6161616161616161616161616161616161616161
```

This issue was considered low-risk as the maximum outcome is a DoS. The fact is that DoS can also be achieved by simply cutting the connection. Still it is recommended to verify the return value from *gmtime()*.

CHR-01-002 General: Wrappers around malloc() do not check for Overflows *(Low)*

While searching for code paths that could possibly result in general problems, it was noticed that Chrony's wrappers around *malloc()* and *realloc()* are prone to potential integer overflows when they are used with their implementation for arrays. This poses an unnecessary risk for relying on those functions without doing manual overflow checks beforehand. The snippet supplied next demonstrates the relevant code.

Affected File:
/chrony/memory.h

Affected Code:
```
#define MallocNew(T) ((T *) Malloc(sizeof(T)))
#define MallocArray(T, n) ((T *) Malloc((n) * sizeof(T)))
#define ReallocArray(T,n,x) ((T *) Realloc((void *)(x), (n)*sizeof(T)))
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Having a multiplication operation can quickly result in scenarios of the combined parameters overflow. This takes place before they are passed to *malloc()* and thus potentially allocates less memory than it was intended. One such problematic code part can be consulted below.

Affected File:
/chrony/ntp_sources.c

Affected Code:

```
static void
rehash_records(void)
{
[...]
  unsigned int i, old_size, new_size;
[...]

  old_size = ARR_GetSize(records);

  temp_records = MallocArray(SourceRecord, old_size);
  memcpy(temp_records, ARR_GetElements(records), old_size * sizeof
(SourceRecord));
[...]
  for (i = 0; i < old_size; i++) {
    if (!temp_records[i].remote_addr)
      continue;

    find_slot(temp_records[i].remote_addr, &slot, &found);
    assert(!found);

    *get_record(slot) = temp_records[i];
  }
```

Depending on the size of *old_size,* the multiplication with *sizeof(SourceRecord)* can result in a zero-sized allocation, which is also due to *Malloc()* not checking for a zero parameter. What is more, it can later lead to an out-of-bounds access through *tmp_records*. This would not happen if the macros around *Malloc()* utilized GCC's special functionalities like *__builtin_mul_overflow,* as described in the GCC manual[1]. The provided *calloc()* example perfectly illustrate a similar use case within the same scenario. It is recommended to implement appropriately corresponding overflow checks for *MallocArray()* and *ReallocArray()*.

[1] https://gcc.gnu.org/gcc-5/changes.html#c-family

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Conclusions

The overwhelmingly positive result of this security assignment performed by three Cure53 testers can be clearly inferred from a marginal number and low-risk nature of the findings amassed in this report.

Withstanding eleven full days of on-remote testing in August of 2017 means that Chrony is robust, strong, and developed with security in mind. The software boasts sound design and is secure across all tested areas. It is quite safe to assume that untested software in the Chrony family is of a similarly exceptional quality.

In general, the software proved to be well-structured and marked by the right abstractions at the appropriate locations. While the functional scope of the software is quite wide, the actual implementation is surprisingly elegant and of a minimal and just necessary complexity. In sum, the Chrony NTP software stands solid and can be seen as trustworthy, especially in comparison to other NTP distributions investigated in recent past.

Cure53 would like to thank Gervase Markham of Mozilla, Miroslav Lichvar of Red Hat and the Chrony team for their excellent project coordination, support and assistance, both before and during this assignment.