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 NTP 01.2017

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

Index

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

Introduction

> *"NTP is a protocol designed to synchronize the clocks of computers over a network. NTP version 4, a significant revision of the previous NTP standard, is the current development version. It is formalized by RFCs released by the IETF."*
>
> From http://www.ntp.org/

This report documents the findings of a source code audit of the NTP software. The project was completed by Cure53 team in January 2017. Four members of the Cure53 participated in this assignment, which required a total of thirty-two days of testing in order for a satisfactory level of coverage to be reached.

The audit constituted a joint project dedicated to both NTP and NTPsec. The code base of NTPsec was examined in parallel and the two components of the scope were given the same amount of attention and scrutiny. While this document primarily pertains to the NTP element, the relevant results applicable to NTPsec are also briefly recalled. At the same time, a separate report has been created to discuss the NTPsec issues in detail and at length. In the latter NTPsec document, the discoveries connected to NTP are analogically given less space and specificity in reporting.

As for the test's approach, the investigations were rooted in the so-called white-box methodology, meaning that the testing team was granted full access to relevant sources and the like. Prior to initiating the audit, the Cure53 team established solid communication channels for the two respective software items in scope, liaising with the development teams of NTP and NTPsec, respectively.

The document proceeds with describing the test's scope, then discusses coverage and findings, ultimately delivering conclusions and verdicts about the general level of security discovered, and the state of the audited code for the two software products in question.

Scope

- **NTP 4.2.8.p9**
  - https://www.eecis.udel.edu/~ntp/ntp_spool/ntp4/ntp-4.2/ntp-4.2.8p9.tar.gz

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

Test Coverage

One can consult an overview of the test's coverage below. The listing shows the percentage of the coverage reached by the code audit per directory. All directories belong to the downloaded code base.

Coverage in %	SLOC	Directory
n/a	88991	sntp
100	68098	ntpd
20	39912	lib
20	12637	libntp
n/a	8791	tests
100	7626	include
100	7578	ports
80	7452	ntpq
n/a	7321	util
n/a	5934	scripts
n/a	5720	libparse
n/a	5148	ntpdc
100	1565	ntpdate
n/a	1520	parseutil
n/a	1422	ntpsnmpd
n/a	950	libjsmn
n/a	934	clockstuff
80	730	kernel
n/a	353	adjtimed

In addition to the directories marked with "*n/a*", the obsolete subcomponent autokey was explicitly left out of scope along with the external libraries (*libevent*, *libisc*, *libopts*), and the testing framework unity.

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

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. *NTP-01-001*) for the purpose of facilitating any future follow-up correspondence.

NTP-01-002 NTP: Buffer Overflow in *ntpq* when fetching *reslist* (*Critical*)

Note: This issue affects NTP only and is not present in the NTPsec code.

A stack buffer overflow can be triggered by a malicious server when a client (using *ntpq*) requests the restriction list from the server. This is due to a missing *length* check in the *reslist()* function. It occurs whenever the function parses the server's response and encounters a *flagstr* variable of an extensive *length*. The string will be copied into a fixed-size buffer, leading to an overflow on the function's stack-frame.

Although this issue is mitigated by having the *ntpq* compiled with *FORTIFY_SOURCE,* the default compilation flags do not include this option (see NTP-01-001), thus leaving a considerable percentage of clients vulnerable. Additionally, stack canaries do not entirely prevent this issue since the overflow allows to overwrite pointers which are used for reading and writing from and to memory before the stack canary is reached. Having control over these pointers makes it possible for an attacker to bypass the stack canary protection.

Affected File:
ntp/ntpq/ntpq-subs.c

Affected Code:

```
typedef struct reslist_row_tag {
       u_int        idx;
       sockaddr_u   addr;
       sockaddr_u   mask;
       u_long       hits;
       char         flagstr[128];
} reslist_row;

static void
reslist(
       struct parse *      pcmd,
       FILE *       fp
```

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

```
        )
{
[...]
        reslist_row   row;
[...]
        qres = doquery(CTL_OP_READ_ORDLIST_A, 0, TRUE, qdata_chars,
                       qdata, &rstatus, &dsize, &datap);
[...]
        ZERO(row);
        fields = 0;
        ui = 0;
        while (nextvar(&dsize, &datap, &tag, &val)) {
[...]
                case 'f':
                        if (1 == sscanf(tag, flags_fmt, &ui)) {
                                if (NULL == val) {
                                        row.flagstr[0] = '\0';
                                        comprende = TRUE;
                                } else {
                                        len = strlen(val);
                                        memcpy(row.flagstr, val, len);
                                        row.flagstr[len] = '\0';
                                        comprende = TRUE;
                                }
                        }
                        break;
```

To verify this issue, the NTP server component was modified to replace every *flagstr* variable with a string of 312-bytes in length prior to sending the response to the client.

Modified File:
ntp/ntpd/ntp_control.c

Modified Code:

```
static void
send_restrict_entry(
        restrict_u *  pres,
        int           ipv6,
        u_int         idx
        )
{
[...]
                case 3:
                        snprintf(tag, sizeof(tag), flags_fmt, idx);
                        match_str = res_match_flags(pres->mflags);
                        access_str = res_access_flags(pres->flags);
                        if ('\0' == match_str[0]) {
```

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

```
        pch = access_str;
} else {
        LIB_GETBUF(buf);
        snprintf(buf, LIB_BUFLENGTH, "%s %s",
                match_str, access_str);
        pch = buf;
}
// TODO: start - patch to simulate malicious server
char payload[1024] = {0};
memset(payload, 'A', 312);
ctl_putunqstr(tag, payload, strlen(payload));
// TODO: original code
// ctl_putunqstr(tag, pch, strlen(pch));
// TODO: end

break;
```

Once the client connects to the malicious server and requests the restriction list, the server responds with a modified package carrying the *flagstr* to trigger the overflow. As soon as the client tries to parse the response, the overflow is triggered and the application crashes.

This behavior can be observed in the following command line output.

Client Output:

```
$ ./ntpq/ntpq localhost
ntpq> reslist
Keyid: 1
MD5 Password:
   hits    addr/prefix or addr mask
           restrictions
===============================================================================
[1]    8995 segmentation fault  ./ntpq/ntpq localhost
```

Observing the crash using GDB, it is clear that the segmentation fault gets triggered by trying to write to an unmapped memory region. This is due to the previously mentioned overwritten pointers. In this case the *RAX* register is controlled by the attacker. However, looking at the stack frame, it is also clear that the saved instruction pointer was overwritten with the malicious flag string.

GDB Output:

```
[-------------------------------------code-------------------------------------]
   0x41016a <reslist+646>: lea    rdx,[rbp-0x130]
   0x410171 <reslist+653>: mov    rax,QWORD PTR [rbp-0x18]
   0x410175 <reslist+657>: add    rax,rdx
```

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

```
=> 0x410178 <reslist+660>: mov    BYTE PTR [rax+0x8],0x0
   0x41017c <reslist+664>: mov    DWORD PTR [rbp-0x4],0x1
   0x410183 <reslist+671>: jmp    0x41023a <reslist+854>
   0x410188 <reslist+676>: jmp    0x41023a <reslist+854>
   0x41018d <reslist+681>: mov    rax,QWORD PTR [rbp-0x90]
[-------------------------------------stack--------------------------------------]
0000| 0x7fffffffdb90 --> 0x7ffff7060780 --> 0xfbad2a84
0008| 0x7fffffffdb98 --> 0x7fffffffdd40 --> 0x642840 --> 0x7473696c736572
('reslist')
0016| 0x7fffffffdba0 --> 0x7b00000200000000
0024| 0x7fffffffdba8 --> 0x25bca8c0
0032| 0x7fffffffdbb0 --> 0x0
0040| 0x7fffffffdbb8 --> 0x0
0048| 0x7fffffffdbc0 --> 0xffffffff7b000002
0056| 0x7fffffffdbc8 --> 0x0
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000410178 in reslist ()
gdb-peda$ i f
Stack level 0, frame at 0x7fffffffdd20:
 rip = 0x410178 in reslist; saved rip = 0x4141414141414141
 called by frame at 0x7fffffffdd28
 Arglist at 0x7fffffffdd10, args:
 Locals at 0x7fffffffdd10, Previous frame's sp is 0x7fffffffdd20
 Saved registers:
  rbp at 0x7fffffffdd10, rip at 0x7fffffffdd18
gdb-peda$
```

To fix the vulnerable code path, it is important to ensure that the length of the flag string does not exceed the buffer size of 128 bytes. Since *memcpy()* is used for the copy operation, a proper *null* termination of the copied string is required.

NTP-01-012 NTP: Authenticated DoS via Malicious Config Option *(High)*

Note: This issue affects both NTP and NTPsec and is present in both code bases.

A vulnerability found in the NTP server allows an authenticated remote attacker to crash the daemon by sending an invalid setting via the *:config* function. The "*unpeer*" option expects a number or an address as an argument. In case the value is "0", a segmentation fault occurs. An example is given in the following listing.

Configuring the server remotely:

```
ntpq> :config unpeer 0
Keyid: 1
MD5 Password:
```

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

```
localhost: timed out, nothing received
***Request timed out
```

The submission of the configuration crashes the NTP server right away. An observation conducted with the GDB shows that the error occurs in *ntp_config.c* and is due to a *null* pointer dereference.

Segmentation fault:

```
(gdb) r
[...]
Program received signal SIGSEGV, Segmentation fault.
0x000000000040b8e5 in config_unpeers (ptree=<optimized out>) at
ntp_config.c:4059
4059                AF(&peeraddr) = curr_unpeer->addr->type;
(gdb) p *curr_unpeer
$3 = {link = 0x0, assocID = 0, addr = 0x0}
```

The *unpeer* configuration options are processed in the *config_unpeers()* function in *ntpd/ntp_config.c*. The *curr_unpeer* struct contains the provided parameter which is either a number or an address. While *assocID* holds numeric values, the *addr* is a pointer to another struct (in case the parameter is an address). If the *curr_unpeer->assocID* is zero, then the code expects *curr_unpeer->addr.* However, the latter is not necessarily the case. Setting "*unpeer 0*" leads to a completely empty *curr_unpeer* struct and thereby crashes the server.

Affected File:
ntp/ntpd/ntp_config.c

Affected Code:

```
config_unpeers(
       config_tree *ptree
       )
{
[...]
       curr_unpeer = HEAD_PFIFO(ptree->unpeers);
       for (; curr_unpeer != NULL; curr_unpeer = curr_unpeer->link) {
[...]
              if (curr_unpeer->assocID) {
[...]
                     continue;
              }

              ZERO(peeraddr);
              AF(&peeraddr) = curr_unpeer->addr->type;
```

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

This issue can be addressed by implementing an alternative verification-handling for the *addr* element in case the *assocID* is zero.

NTP-01-015 NTPsec: Regression in *ctl_putdata()* leads to Endless Loop *(High)*

Note: This issue affects NTPsec only and is a regression from a security fix.

The original vulnerability tracked under *CVE-2014-9295* was fixed by an initial patch[1] on the 12th of December, 2014. However, the code in question appears to have undergone changes and revisions on or around November 4th, 2016, which removed the fix. These can be seen in a subsequent *commit*[2] from that period.

NTP-01-016 NTP: Denial of Service via Malformed Config *(High)*

Note: This issue affects both NTP and NTPsec and is present in both code bases.

A vulnerability found in the NTP server makes it possible for an authenticated remote user to crash the service via a malformed configuration. After submitting the *config* line in the following snippet, the *ntp daemon* crashes after a couple of seconds.

Configuring the server remotely:

```
ntpq> :config server 10.0.0.1 mode 3735928559
Keyid: 1
MD5 Password:
Config Succeeded
```

The exact reason for the crash can be derived from running the daemon in the GDB. Here an invalid value in the *%rax* register leads to an invalid *read* operation and causes a segmentation fault.

Segmentation fault:

```
(gdb) r
[...]
Program received signal SIGSEGV, Segmentation fault.
0x0000000000429417 in peer_xmit (peer=0x6a7ee0 <init_peer_alloc+32>,
peer@entry=0x6a7ec0 <init_peer_alloc>) at ntp_proto.c:3722
3722          sendpkt(&peer->srcadr, peer->dstadr, sys_ttl[peer->ttl],
(gdb) x/i $rip
=> 0x429417 <peer_xmit+1367>:    movzbl 0x6f7018(%rax),%edx
(gdb) p/x $rax
$1 = 0xdeadbeef
```

[1] https://github.com/ntpsec/ntpsec/commit/7fd82020dfd501ee4510edbd61eaf1eb796d5db9
[2] https://github.com/ntpsec/ntpsec/commit/1a545205529b17390a7ae93bfc069b5a517c95bc

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

The affected code is in the *ntp_proto.c* file of the *peer_xmit()* function. *Peer* is a struct which contains several values, including the user-controlled *peer->ttl* variable. An invalid value causes an invalid memory access.

Affected File:
ntp/ntpd/ntp_proto.c

Affected Code:
```
peer_xmit(
       struct peer *peer   /* peer structure pointer */
       )
[...]
       sendpkt(&peer->srcadr, peer->dstadr, sys_ttl[peer->ttl],
              &xpkt, sendlen);
```

The core problem resides within the configuration parser where the parameters from the configuration lines are stored in node structs. Setting the *ttl* value can be done in two different ways ("*ttl*" and "*mode*") but only one of them checks the provided number for sanity. While invalid numbers are being ignored when "*ttl*" option is used, no checks are performed for "*mode*".

Affected File:
ntp/ntpd/ntp_config.c

Affected Code:
```
peer_node *
create_peer_node(
       int           hmode,
       address_node *      addr,
       attr_val_fifo *     options
       )
       [...]
       case T_Ttl:
              if (option->value.u >= MAX_TTL) {
                     msyslog(LOG_ERR, "ttl: invalid argument");
                     errflag = 1;
              } else {
                     my_node->ttl = (u_char)option->value.u;
              }
              break;

       case T_Mode:
              my_node->ttl = option->value.u;
              break;
```

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

It was not possible to exploit this issue beyond achieving DoS. However, it is recommended to resolve this problem by adding sanity checks to the *subtype* and *mode* configuration options.

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.

NTP-01-001 NTP: Makefile does not enforce Security Flags *(Low)*

Note: This issue affects both NTP and NTPsec and is present in both code bases.

One of the key realms reviewed quite early during almost every security test of a new project encompasses studying the presence of hardening flags applied when the software is built. This can be done with tools like *checksec*[3] or *PEDA*[4] once the software has been compiled with the default options inside the *makefile* at hand:

```
$ gdb ./ntpd
Reading symbols from ./ntpd...done.
(gdb) checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : Partial
(gdb)
```

From the GDB's output it is apparent that the hardening flags are derived from the global Linux distribution setting rather than forced from the *makefile* itself. From this follows that certain hardening checks are missing. These include stack canaries, which ordinarily protect the *return address* from buffer overflow vulnerabilities on the stack, as well as *FORTIFY_SOURCE.*

It is important to set the necessary *CFLAGS* inside the *makefile* itself in order to directly instruct the compiler to insert all of the security flags required. Once activated, the exploitation of multiple kinds of memory corruption vulnerabilities becomes much more difficult. This increase in security stems from two reasons: one having to do with

[3] http://www.trapkit.de/tools/checksec.html
[4] https://github.com/longld/peda

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

requiring additional information leaks from the program's memory, and the other revolving around establishing that the problems are mitigated by, for example, newly introduced *length* checks.

The following snippet shows what *CFLAGS* are recommended for an addition to the *make* process:

```
$ make CFLAGS='-Wl,-z,relro,-z,now -pie -fPIE -fstack-protector-all
-D_FORTIFY_SOURCE=2 -O1'
[...]
$ gdb ./ntpd
Reading symbols from ./ntpd...done.
(gdb) checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
(gdb)
```

NTP-01-003 NTP: Improper use of *snprintf()* in *mx4200_send()* *(Low)*

Note: This issue affects both NTP and NTPsec and is present in both code bases.

The function *mx4200_send()* uses the *libc* function *snprintf()/vsnprintf()* incorrectly. This can lead to an out-of-bounds memory write due to an improper handling of the return value of *snprintf()/vsnprintf()*. Said value returns the number of bytes it would have written if there were no length restrictions in place.

The code in question takes the return value outlined above and increments an iterator by its value. This iterator is supposed to point into the fixed-size buffer. However, since the return value can be larger than the buffer's size, it is possible for the iterator to point somewhere outside of the allocated buffer space. This results in an out-of-bound memory *write* in the *snprintf()* specified in this ticket. The reason behind the problem is that the iterator is used as the destination pointer.

This behavior can be leveraged to overwrite a saved instruction pointer on the stack and gain control over the execution flow. During the test it was not possible to identify any malicious usage for this function, specifically no way for an attacker to exploit the issue mentioned above was ultimately unveiled. However, this remains to be a problem capable of introducing new vulnerabilities. The problems are likely to resurface when new code that uses this function is added. In other words, it is necessary to fix this flaw in advance.

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

Affected File:
ntp/ntpd/refclock_mx4200.c

Affected Code:

```
#if defined(__STDC__)
static void
mx4200_send(struct peer *peer, char *fmt, ...)
#else
static void
mx4200_send(peer, fmt, va_alist)
     struct peer *peer;
     char *fmt;
     va_dcl
#endif /* __STDC__ */
{
[...]
char buf[1024];
[...]

       cp = buf;
       *cp++ = '$';
       n = VSNPRINTF((cp, sizeof(buf) - 1, fmt, ap));
       ck = mx4200_cksum(cp, n);
       cp += n;
       ++n;
       n += SNPRINTF((cp, sizeof(buf) - n - 5, "*%02X\r\n", ck));
```

In the above code, *cp* initially points to the beginning of the buffer. Once the *vsnprintf()* returns, *mx4200_cksum()* is called for creating a checksum. This is done by iterating over each byte of the buffer. However, the *mx4200_cksum()* uses the return value *n* from *vnsprintf()* to determine the length of the buffer it needs to iterate over. Since *n* may be larger than the buffer's size, an out-of-bounds read can occur as a result of creating the checksum.

It is recommended to check the return value of the *vsnprintf()/snprintf()* and ensure that it does not exceed the size allowed for a buffer. Also, before calling *snprintf()*, it must be ensured that at least five bytes are available, with the view to avoiding an overflow.

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

NTP-01-004 NTP: Potential Overflows in *ctl_put()* functions *(Medium)*

Note: This issue affects both NTP and NTPsec and is present in both code bases.

For the purpose of formatting different kinds of response strings into each response packet, *Ntpd* makes use of different wrappers around *ctl_putdata()*. For example, *ctl_putstr()* is often used to send quoted system variables, while *ctl_putuint()* comes into the fore when integer responses are being handled. All of these wrappers, however, suffer from stack based buffer overflow vulnerabilities as soon as they are utilized incorrectly. This is due to the fact that the length of the source variable is used on each occasion when the data is being copied into a local buffer. This is highlighted in the provided code.

Affected File:
ntp/ntpd/ntp_control.c

Affected Code:

```
ctl_putstr(
       const char *  tag,
       const char *  data,
       size_t        len
       )
{
       char buffer[512];
       char *cp;
       size_t tl;

       tl = strlen(tag);
       memcpy(buffer, tag, tl);
```

While this issue should be considered hard to exploit with the presence of the stack canaries and is actually mitigated by *FORTIFY_SOURCE*, these functions nevertheless pose a considerable threat as soon as they operate on values larger than the destination size.

Although the current state of NTP appears not to permit setting tag lengths greater than 512 bytes (mainly because they all have static values), it is still recommended to fix all *ctl_put* functions by limiting the source length.

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

NTP-01-005 NTP: Off-by-one in Oncore GPS Receiver *(Low)*

Note: This issue affects both NTP and NTPsec and is present in both code bases.

Regardless of bugs inside the *refclock* drivers not posing high security risks, the Cure53 testing team discovered several coding errors worth reporting. One mistake was found in the Oncore GPS Receiver of Motorola devices. The vulnerable code can be found below.

Affected File:
ntp/ntpd/refclock_oncore.c

Affected code:

```
static void
oncore_receive(
       struct recvbuf *rbufp
       )
{
       size_t i;
       u_char *p;
       struct peer *peer;
       struct instance *instance;

       peer = rbufp->recv_peer;
       instance = peer->procptr->unitptr;
       p = (u_char *) &rbufp->recv_space;
[...]
       i = rbufp->recv_length;
       if (rcvbuf+rcvptr+i > &rcvbuf[sizeof rcvbuf])
              i = sizeof(rcvbuf) - rcvptr;     /* and some char will be lost */
       memcpy(rcvbuf+rcvptr, p, i);
       rcvptr += i;
       oncore_consume(instance);
}
```

The highlighted length check above incorrectly sets the boundaries for the received buffer by limiting to *sizeof(rcvbuf).* In this context, an alternative *sizeof(rcvbuf) - 1* would be correct because the size is used as an index. This creates an *off-by-one* buffer overflow. Since *rcvbuf* is directly followed by another buffer, this issue is deemed nearly impossible to exploit. Still, it should be viewed as a coding error and resolved accordingly.

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

NTP-01-006 NTP: Copious amounts of Unused Code *(Info)*

Note: This issue affects NTP only and is not present in the NTPsec code.

Statically included external projects potentially introduce several problems and the issue of having extensive amounts of code that is “dead” in the resulting binary must clearly be pointed out. The unnecessary unused code may or may not contain bugs and, quite possibly, might be leveraged for code-gadget-based branch-flow redirection exploits. Analogically, having source trees statically included as well means a failure in taking advantage of the free feature for periodical updates. This solution is offered by the system’s Package Manager.

Affected Directories:
ntp/lib/isc/
ntp/sntp/libevent/
ntp/sntp/libopts/

It is recommend for the static external libraries to be removed and replaced with only the small fragments of the actually used code. Alternatively, libraries should be, at the very minimum, introduced as external dependencies, meaning that future updates and patches are included automatically.

NTP-01-007 NTP: Data Structure terminated insufficiently *(Low)*

Note: This issue affects NTP only and is not present in the NTPsec code.

Calling *strcpy()* with an argument of *string* with additional *null* bytes actually only copies a single terminating *null* character into the target buffer instead of relying on the required *double null* bytes in *addKeysToRegistry()* function. As a consequence, a garbage registry entry can be created and consist leaked memory contents. The additional *arsize* parameter is erroneously set to contain two *null* bytes and the following call to *RegSetValueEx()* claims to be passing in a multi-string value, though this cannot be guaranteed.

Affected File:
ntp/ports/winnt/instsrv/instsrv.c

Affected Code:
```
int addKeysToRegistry()
{
[...]
  char   myarray[200];
```

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

```
  char *lpmyarray = myarray;
  int arsize = 0;

[...]
  strcpy(lpmyarray,"TcpIp");
  lpmyarray = lpmyarray + 6;
  arsize = arsize + 6;
  strcpy(lpmyarray,"Afd");
  lpmyarray = lpmyarray + 4;
  arsize = arsize + 4;
  arsize = arsize + 2;
  strcpy(lpmyarray,"\0\0");

  bSuccess = RegSetValueEx(hk,  /* subkey handle         */
      "DependOnService",        /* value name           */
      0,                        /* must be zero         */
      REG_MULTI_SZ,             /* value type           */
      (LPBYTE) &myarray,        /* address of value data */
      arsize);                  /* length of value data  */
```

NTP-01-008 NTP: Stack Buffer Overflow from Command Line *(Low)*

Note: This issue affects NTP only and is not present in the NTPsec code.

Invoking *strcat()* blindly appends the string passed to stack buffer in the *addSourceToRegistry()* function. The stack buffer is 70 bytes smaller than the buffer in the calling *main()* function. Together with the initially copied *Registry* path, the combination causes a stack buffer overflow and effectively overwrites the stack frame. The passed application path is actually limited to 256 bytes by the operating system, but this is not sufficient to assure that the affected stack buffer is consistently protected against overflowing at all times.

Affected File:
ntp/ports/winnt/instsrv/instsrv.c

Affected Code:

```
int addSourceToRegistry(LPSTR pszAppname, LPSTR pszMsgDLL)
{
[...]
  char   regarray[200];

char *lpregarray = regarray;

[...]
  strcpy(lpregarray,
         "SYSTEM\\CurrentControlSet\\Services\\EventLog\\Application\\");
```

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

```
strcat(lpregarray, pszAppname);
```

It is recommend for the respective size-limited and properly terminating library functions being used instead. More specifically, *strlcpy()* and *strlcat()* should replace the obsolete and error-prone solutions currently in place. Conversely, it is not a desirable approach to simply adjust the buffer size to accommodate the size of the buffer passed from the caller.

NTP-01-009 NTP: Privileged execution of User Library code *(Low)*

Note: This issue affects NTP only and is not present in the NTPsec code.

The Windows NT port has the added capability to preload DLLs defined in the inherited global local environment variable *PPSAPI_DLLS.* The code contained within those libraries is then called from the NTPD service, usually running with elevated privileges. Depending on how securely the setup of the respective machine is configured, this can easily lead to an injection of code similar but not equivalent to *LD_PRELOAD* on the UNIX-like operating systems.

Affected Files:
ntp/ports/winnt/include/timepps.h
ntp/port/winnt/ppsapi/loopback/src/timepps.h

Affected Code:

```
static inline int
time_pps_create(
       int           filedes,/* device file descriptor */
       pps_handle_t *      phandle      /* returned handle */
       )
{
[...]
       char *              dlls;
       char *              dll;
       char *              pch;
[...]
       int                 err;
       dlls = getenv("PPSAPI_DLLS");

       if (dlls != NULL && NULL == g_provider_list) {
              dlls = dll = _strdup(dlls);
              fprintf(stderr, "getenv(PPSAPI_DLLS) gives %s\n", dlls);
       } else
              dlls = dll = NULL;
```

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

```
        while (dll != NULL && dll[0]) {
                pch = strchr(dll, ';');
                if (pch != NULL)
                        *pch = 0;
                err = load_pps_provider(dll);
[...]
                dll = (NULL == pch)
                        ? NULL
                        : pch + 1;
        }
```

Considering the testing team's numerous encounters with poorly maintained Windows servers, it is recommended to eradicate the current mechanism and replace it with a *Registry*-based approach.

NTP-01-010 NTP: *ereallocarray()/eallocarray()* underused (*Info*)

Note: This issue affects NTP only and is not present in the NTPsec code.

NTP makes use of several wrappers around the standard heap memory allocation functions that are provided by *libc*. This is mainly done to introduce additional safety checks concentrated on several goals. First, they seek to ensure that memory is not accidentally freed, secondly they verify that a correct amount is always allocated and, thirdly, that allocation failures are correctly handled. There is an additional implementation for scenarios where memory for a specific amount of items of the same size needs to be allocated. The handling can be found in the *oreallocarray()* function for which a further *number-of-elements* parameter needs to be provided:

File:
ntp/libntp/emalloc.c

Code:

```
#define MUL_NO_OVERFLOW    ((size_t)1 << (sizeof(size_t) * 4))
void *
oreallocarray(
        void *optr,
        size_t nmemb,
        size_t size

[...]
{
        if ((nmemb >= MUL_NO_OVERFLOW || size >= MUL_NO_OVERFLOW) &&
            nmemb > 0 && SIZE_MAX / nmemb < size) {
#ifndef EREALLOC_CALLSITE
                msyslog(LOG_ERR, "fatal allocation size overflow");
```

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

The described function additionally ensures that the later multiplication of *size* * *nmemb* does not create an integer overflow. In other words, it is responsible for attesting to less memory than originally intended not being allocated in a given case. The problem, however, is that the function in question is used quite rarely, even though there are some places calling for it to be employed instead of the usual *emalloc*. An example is supplied next.

File:
ntp/ntpd/ntp_loopfilter.c

Code:
```
sys_huffpuff = emalloc(sizeof(sys_huffpuff[0]) * sys_hufflen);
```

File:
ntp/ntpd/ntp_peer.c

Code:
```
peers = emalloc_zero(INC_PEER_ALLOC * sizeof(*peers));
```

Although no considerable threat was identified as tied to a lack of this function, it is recommended to correctly apply *oreallocarray* as a preferred option across all of the locations where it is possible.

NTP-01-011 NTP: *ntpq_stripquotes()* returns incorrect Value *(Low)*

Note: This issue affects NTP only and is not present in the NTPsec code.

The NTP client (*ntpq*) uses the function *ntpq_stripquotes()* to remove quotes and escape characters from a given string. According to the documentation, the function is supposed to return the number of copied bytes but due to incorrect pointer usage this value is always zero.

Affected File:
ntp/ntpq/libntpq.c

Affected Code:
```
int ntpq_stripquotes ( char *resultbuf, char *srcbuf, int datalen, int maxlen )
{
	char* tmpbuf = srcbuf;
[...]

	*resultbuf = 0;
```

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

```
        return strlen(resultbuf);
}
```

Although the return value of this function is never used in the code, this flaw could lead to a vulnerability in the future. Since relying on wrong return values when performing memory operations is a dangerous practice, it is recommended to return the correct value in accordance with the documentation pertinent to the code.

NTP-01-013 NTPsec: Inclusion of obsolete NTPclassic-dependent Script *(Info)*

Note: This issue affects NTPsec only.

The NTPsec project includes an inapplicable script dependent on the NTPclassic's *ntpq*. This inclusion is believed to be a mere oversight, which can be likely attributed to the challenges of the repository conversion.

NTP-01-014 NTP: Buffer Overflow in DPTS Clock *(Low)*

Note: This issue affects NTP only and is not present in the NTPsec code.

Another potential issue inside the *refclock* drivers was found in the receiver for the Datum Programmable Time Server. Here the packets are processed from the */dev/datum* device and handled in *datum_pts_receive()* in the following code:

Affected File:
ntp/ntpd/refclock_datum.c

Affected Code:

```
static void
datum_pts_receive(
       struct recvbuf *rbufp
       )
{
       int i;
       l_fp tstmp;
       struct peer *p;
       struct datum_pts_unit *datum_pts;
       char *dpt;
       int dpend;
       int tzoff;
       int timerr;
       double ftimerr, abserr;
[...]
```

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

```
        p = rbufp->recv_peer;
        datum_pts = p->procptr->unitptr;
        dpt = (char *)&rbufp->recv_space;
        dpend = rbufp->recv_length;
[...]

        for (i=0; i<dpend; i++) {
                datum_pts->retbuf[datum_pts->nbytes+i] = dpt[i];
        }
```

Since *dpend* simply holds the length of the entire packet, the loop highlighted above will continue the process of copying data into *datum_pts->retbuf*, even though there is only room for 8 bytes there. This is a classic buffer overflow inside a data structure that is stored on the heap.

Since an attacker would be required to somehow control a malicious */dev/datum* device, this does not appear to be a practical attack and renders this issue “Low” in terms of severity. To be on the safe side, however, it is still recommended to fix the overflow by limiting the amount of the incoming data to match the size of the destination buffer.

Conclusion

The joint nature of this January 2017 code audit, performed by the Cure53 team against both the NTP and the NTPsec software components in scope, makes it that much complex to issue an unambiguous verdict about all security-relevant aspects.

The bottom line is that four members of the Cure53 team, who assessed the products over the course or thirty-two days, discovered sixteen individual findings in the code base of both NTP and NTPsec. Furthermore, one finding concerning NTP was flagged with a “Critical” severity ranking due to its high-impact implications. Breaking down the findings moreover indicates that eight of the discoveries were exclusively tied to NTP entity, while other two could be linked exclusively to the realm of NTPsec. This means that six spotted problems were shared between the two code bases. In other words, the total numbers of findings suggest fourteen issues for NTP and eight for NTPsec.

The general outcome of this project is rooted in the fact that the code has been left to grow organically and had aged somewhat unattended over the years. The overall structure has thus become very intricate, while also yielding a conviction that different styles and approaches were used and subsequently altered. The seemingly uncontrolled inclusion of variant code via header files and complete external projects engenders a particular problem. Most likely, it makes the continuous development much more difficult than necessary.

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

In sum, the maintainers are encouraged to engage in a meticulously organized process of cleaning up the code base, removing unnecessary cruft, and eradicating any obsolete subcomponents. While the state of security is evidently not optimal, there is definite room for growth, code stability and overall security improvement as long as more time and efforts are invested into the matter.

Cure53 would like to thank Gervase Markham of Mozilla for his excellent project coordination, support and assistance, both before and during this assignment. Cure53 would further like to extend gratitude to the NTP team, for their help during the scoping phase of this assessment.