April 21, 2020

GHSL-2020-051, GHSL-2020-052: Multiple vulnerabilities in NTOP nDPI


[Bas Alberts](#)

Summary

The [NTOP](#) Deep Packet Inspection Toolkit is driven in large part by the nDPI library. This library contains a large set of network protocol dissectors intended to parse and analyze packet-captured network traffic.

The nDPI SSH protocol dissector suffers from multiple integer overflow vulnerabilities which result in a controlled remote heap overflow. Due to the granular nature of the overflow primitive and the ability to control both the contents and layout of the nDPI library's heap memory through remote input, this vulnerability may be abused to achieve full Remote Code Execution (RCE) against any network inspection stack that is linked against nDPI and uses it to perform network traffic analysis.

The nDPI SSH protocol dissector also suffers from an Out Of Bounds (OOB) read vulnerability which may result in a Denial of Service (DoS).

These specific vulnerabilities were introduced in the 3.0-stable release of nDPI.

CVE

- CVE-2020-11939
- CVE-2020-11940

Product

NTOP nDPI (https://github.com/ntop/nDPI)

Tested Version

The development branch of nDPI as of Mar 23, 2020 and commit https://github.com/ntop/nDPI/commit/c6acf97bfbe5ad26db3c2f5dd4d379ac674d6fb3#diff-a3a2b66d47ec1a3eab1b650f55f68ab7

Details

Multiple integer overflows in `ssh.c:concat_hash_string` (GHSL-2020-051, CVE-2020-11939)

NOTE: annotated source code based on: https://github.com/ntop/nDPI/blob/c6acf97bfbe5ad26db3c2f5dd4d379ac674d6fb3/src/lib/protocols/ssh.c

When dissecting the SSH protocol the nDPI library will actively parse KEXINIT (type: 20) messages in both the server-to-client and client-to-server directions and attempts to extract various descriptive string sets from SSH KEXINIT packets. These string sets include e.g. the key exchange algorithms supported by the client and the server.

The SSH protocol handles string data based on the common `[length][data]` format, where length is a 32bit integer value. As such nDPI extracts these length values, and uses them to pull the actual string data from the SSH KEXINIT packets. nDPI also uses these length values to calculate and maintain a running offset in the captured packet data. This offset is maintained as a 16bit unsigned integer variable.

For example, to pull the key exchange algorithms out of a SSH KEXINIT packet, nDPI performs the following operations:

```
ssh.c:ndpi_search_ssh_tcp
...
    if(msgcode == 20 /* key exchange init */) {
      char *hassh_buf = calloc(packet->payload_packet_len, sizeof(char));
...
          len = concat_hash_string(packet, hassh_buf, 0 /* server */);
...

ssh.c:concat_hash_string
...
  u_int16_t offset = 22, buf_out_len = 0;

  if(offset+sizeof(u_int32_t) >= packet->payload_packet_len)
    goto invalid_payload;

  u_int32_t len = ntohl(*(u_int32_t*)&packet->payload[offset]);
  offset += 4;

  /* -1 for ';' */
  if((offset >= packet->payload_packet_len) || (len >= packet->payload_packet_len-offset-1))
    goto invalid_payload;

  /* ssh.kex_algorithms [C/S] */
  strncpy(buf, (const char *)&packet->payload[offset], buf_out_len = len);
  buf[buf_out_len++] = ';';
  offset += len;
```

We initially make 2 observations. First, that the destination buffer `buf` was previously allocated with a `calloc` call that is sized according to `packet->payload_packet_len`, which represents the actual size of the captured SSH packet. Second, that nDPI attempts to make sure that the `offset` and subsequently the `len` variables do not result in data accesses beyond `packet->payload_packet_len`. The intent here is to prevent read or write access outside of the bounds of the allocated `buf` memory region.

However, when examining the `concat_hash_string` function in greater detail, we note the following pattern:

```
ssh.c:concat_hash_string
...
[1]
  /* ssh.encryption_algorithms_client_to_server [C] */
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);

  if(client_hash) {
    offset += 4;

    if((offset >= packet->payload_packet_len) || (len >= packet->payload_packet_len-offset-1))
      goto invalid_payload;

    strncpy(&buf[buf_out_len], (const char *)&packet->payload[offset], len);
    buf_out_len += len;
    buf[buf_out_len++] = ';';
    offset += len;
  } else
[2]
    offset += 4 + len;

  /* ssh.encryption_algorithms_server_to_client [S] */
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);

  if(!client_hash) {
    offset += 4;

    if((offset >= packet->payload_packet_len) || (len >= packet->payload_packet_len-offset-1))
      goto invalid_payload;
```

```
    strncpy(&buf[buf_out_len], (const char *)&packet->payload[offset], len);
    buf_out_len += len;
    buf[buf_out_len++] = ';';
    offset += len;
  } else
    offset += 4 + len;

[3]
  /* ssh.mac_algorithms_client_to_server [C] */
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);

  if(client_hash) {
    offset += 4;

[4]
    if((offset >= packet->payload_packet_len) || (len >= packet->payload_packet_len-offset-1))
      goto invalid_payload;
[5]
    strncpy(&buf[buf_out_len], (const char *)&packet->payload[offset], len);
    buf_out_len += len;
    buf[buf_out_len++] = ';';
    offset += len;
  } else
[6]
    offset += 4 + len;
```

The `client_hash` variable decides whether the packet direction is server-to-client or client-to-server, but since the parsing pattern is the same for either direction it does not matter much for the sake of our discussion. For the sake of convenience we will examine the `!client_hash` case, but the same logic holds for the inverse.

At [1] we see that we have full control over a 32bit unsigned length integer. At [2] we see that `offset` is updated according to this controlled length variable based on an integer addition operation (`offset + 4 + len`). This arithmetic expression expands to 32bit integer values and is then truncated back to a 16bit integer value on the final assignment back into `offset`. This means that, given we have full control of the `len` variable, we can effectively integer wrap `offset` to be any value we wish it to be.

For example, to make `offset` become `n` where `n` is any desired 16bit integer value, we would simply set `len` to `0 - offset - 4 + n`.

The practical implication here is that, given at [4] the `offset` variable is used to ensure no out of bounds accesses occur, and given our ability to effectively set (and RESET) `offset` to any desired 16bit integer value, we can pass the intended bounds checks by resetting `offset` to a small enough value.

This integer overflow is the core issue resulting in what ultimately becomes a controlled remote heap overflow. However, to mold this scenario into a RCE-viable situation, some additional context is required.

We’ve established that we can effectively reset `offset` in-between the `strncpy` operations that copy remote controlled data into `buf`. When we examine the copy operation at [5], we note that that destination offset into `buf` is controlled by `buf_out_len`. We also note that `buf_out_len` is adjusted upward based on the `len` variable, which is the length of our remote controlled string.

For practical exploitation of this issue, the fact that `buf_out_len` is maintained as it’s own independent offset into the destination buffer becomes relevant.

Let’s recap:

- We control `offset` through an integer overflow
- We can repeat a pattern of: `strncpy(&buf[buf_out_len], controlled_data, controlled_len)`, adjust `buf_out_len` up by `controlled_len`, reset `offset` to any desired `n`
- We can not directly overflow `buf` based on this controlled len due to the `controlled_len >= packet->payload_packet_len-offset-1` check

Because `packet->payload_packet_len` controls the `buf` memory allocation size, and is directly influenced by our packet size, the intuition that we can simply pack repeated large strings to trigger an offset overwrap here is less than ideal from an attacker perspective. However, what we CAN do is pack a single string into the KEXINIT SSH packet and then copy that string repeatedly. A single string will result in a `calloc` based on the string’s size + some minor SSH protocol overhead and since `buf_out_len` is not bounds checked at any point, we can reset the `offset` variable to point at that initial string data for each subsequent `strncpy` operation.

In other words, using the `offset` integer overflow primitive, we can trick the nDPI SSH dissector into repeatedly copying the same data into `&buf[buf_out_len]`. Since `buf_out_len` is adjusted based on our controlled string length, and not otherwise sanity checked, we can write out of bounds as early as the 2nd repeat of the `strncpy` operation, pending string size vs protocol overhead (which you fully control).

This then becomes a controlled remote heap overflow primitive. Since the nDPI lib has a direct relationship to remotely controlled input, and it will allocate, deallocate, and populate heap memory in a direct 1:1 relationship to said remotely controlled input, this becomes a viable primitive to achieve remote code execution.

Out of bounds read in `ssh.c:concat_hash_string` (GHSL-2020-052, CVE-2020-11940)

A comparatively minor, but related, repeating issue exists in the form of an OOB read, e.g. when examining the following snippet:

```
ssh.c:concat_hash_string
...
  /* ssh.server_host_key_algorithms [None] */
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);
[1]
  offset += 4 + len;

  /* ssh.encryption_algorithms_client_to_server [C] */
[2]
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);
...
```

We note that at [1], `offset` is calculated per the previously discussed integer arithmetic, which is susceptible to integer overflow. However, beyond that, we also note that at [2] the resulting `offset` value is immediately used as an index into packet data without any bounds check. This may result in an OOB read due to `offset` being fully user controlled.

A very similar pattern is repeated any time a new `offset` is calculated in a path that does not perform an explicit check against the resulting `offset` value, e.g.:

```
  /* ssh.encryption_algorithms_client_to_server [C] */
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);

  if(client_hash) {
    offset += 4;

    if((offset >= packet->payload_packet_len) || (len >= packet->payload_packet_len-offset-1))
      goto invalid_payload;

    strncpy(&buf[buf_out_len], (const char *)&packet->payload[offset], len);
    buf_out_len += len;
    buf[buf_out_len++] = ';';
    offset += len;
  } else
[1]
    offset += 4 + len;

  /* ssh.encryption_algorithms_server_to_client [S] */
[2]
  len = ntohl(*(u_int32_t*)&packet->payload[offset]);
```

Again, at [1] the initial remote controlled `len` variable is used to set `offset`, immediately after at [2] the resulting `offset` is used as an index into packet data without any bounds check. This may result in an OOB read due to `offset` being fully user controlled.

Impact

These issues may lead to `Remote Code Execution` in the case of `GHSL-2020-051` and `Denial of Service` in the case of `GHSL-2020-052`.

Remediation

These issues were addressed in the following commit:

https://github.com/ntop/nDPI/commit/c120cca66272646c4277d71fa769d020b1026b28

Coordinated Disclosure Timeline

This report was subject to the GHSL [coordinated disclosure policy](#).

- 03/23/2020: initial report sent to ntop team

- 03/24/2020: maintainer acknowledges report receipt and begins triage process
- 03/30/2020: additional findings sent to maintainer
- 04/05/2020: maintainer merges fixes for initial findings
- 04/15/2020: maintainer merges fixes for additional findings
- 04/21/2020: public advisory
- 04/29/2020: additional integer overflow for 32bit systems reported by @rhuizer
- 04/30/2020: revised patch committed by maintainer

Credit

This issue was discovered and reported by GHSL team member @anticomputer (Bas Alberts).

An additional integer overflow issue affecting 32bit platforms was reported by @rhuizer (Ronald Huizer).

Contact

You can contact the GHSL team at `securitylab@github.com`, please include `GHSL-2020-051` or `GHSL-2020-052` in any communication regarding this issue.