SIGRed – Resolving Your Way into Domain Admin: Exploiting a 17 Year-old Bug in Windows DNS Servers

July 14, 2020

Research by: Sagi Tzadik

Introduction

DNS, which is often described as the “phonebook of the internet”, is a network protocol for translating human-friendly computer hostnames into IP addresses. Because it is such a core component of the internet, there are many solutions and implementations of DNS servers out there, but only a few are extensively used.

“Windows DNS Server” is the Microsoft implementation and is an essential part of and a requirement for a Windows Domain environment.

SIGRed (CVE-2020-1350) is a wormable, critical vulnerability (CVSS base score of 10.0) in the Windows DNS server that affects Windows Server versions 2003 to 2019, and can be triggered by a malicious DNS response. As the service is running in elevated privileges (SYSTEM), if exploited successfully, an attacker is granted Domain Administrator rights, effectively compromising the entire corporate infrastructure.

Motivation

Our main goal was to find a vulnerability that would let an attacker compromise a Windows Domain environment, preferably unauthenticated. There is a lot of related research by various independent security researchers as well as those sponsored by nation-states. Most of the published and publicly available materials and exploits focus on Microsoft’s implementation of SMB (EternalBlue) and RDP (BlueKeep) protocols, as these targets affect both servers and endpoints. To obtain Domain Admin privileges, a straightforward approach is to directly exploit the Domain Controller. Therefore, we decided to focus our research on a less publicly explored attack surface that exists primarily on Windows Server and Domain Controllers. Enter WinDNS.

Windows DNS Overview

"*Domain Name System (DNS) is one of the industry-standard suite of protocols that comprise TCP/IP, and together the DNS Client and DNS Server provide computer name-to-IP address mapping name resolution services to computers and users.*" – Microsoft.

DNS primarily uses the User Datagram Protocol (UDP) on port 53 to serve requests. DNS queries consist of a single UDP request from the client followed by a single UDP reply from the server.

In addition to translating names to IP addresses, DNS serves other purposes as well. For example, mail transfer agents use DNS to find the best mail server to deliver e-mail: An MX record provides a mapping between a domain and a mail exchanger, which can provide an additional layer of fault tolerance and load distribution. A list of available DNS record types and their corresponding purposes can be found on Wikipedia.

But the point of this blog post is not to present a lengthy discourse on DNS features and history, so we encourage you to read more about DNS here.

What you need to know:

- DNS operates over UDP/TCP port 53.
- A single DNS message (response / query) is limited to 512 bytes in UDP and 65,535 bytes in TCP.
- DNS is hierarchal and decentralized in nature. This means when a DNS server doesn't know the answer to a query it receives, the query is forwarded to a DNS server above it in the hierarchy. At the top of the hierarchy there are 13 root DNS servers worldwide.

In Windows, the DNS client and DNS server are implemented in two different modules:

- **DNS Client** – `dnsapi.dll` is responsible for DNS resolving.
- **DNS Server** – `dns.exe` is responsible for answering DNS queries on Windows Server, in which the DNS role is installed.

Our research is centered around the `dns.exe` module.

Preparing the Environment

There are two main scenarios for our attack surface:

1. A bug in the way the DNS server parses an incoming query.
2. A bug in the way the DNS server parses a response (answer) for a forwarded query.

As DNS queries do not have a complex structure, there is a lower chance of finding parsing issues in the first scenario, so we decided to target functions that parse incoming responses for forwarded queries.

As mentioned previously, a forwarded query is the utilization of the DNS architecture to be able to forward queries it does not know the answer to – to the DNS server above it in the hierarchy.

However, most environments configure their forwarders to well-known, respectable DNS servers such as `8.8.8.8` (Google) or `1.1.1.1` (Cloudflare), or at the very least a server that is not under the attacker's control.

This means that even if we find an issue in the parsing of DNS responses, we need to establish a Man-in-the-Middle to exploit it. Obviously, that's not good enough.

NS Records to the Rescue

NS stands for 'name server' and this record indicates which DNS server is the authority for that domain (which server contains the actual DNS records). The NS record is usually in charge of resolving the subdomains of a given domain. A domain often has multiple NS records which can indicate primary and backup name servers for that domain.

To have the target Windows DNS Server parse responses from our malicious DNS NameServer, we do the following:

1. Configure our domain's (`deadbeef.fun`) NS Records to point at our malicious DNS Server (`ns1.41414141.club`).
2. Query the victim Windows DNS Server for NS Records of `deadbeef.fun`.
3. The victim DNS, not yet knowing the answer for this query, forwards the query to the DNS server above it (`8.8.8.8`).
4. The authoritative server (`8.8.8.8`) knows the answer, and responds that the NameServer of `deadbeef.fun` is `ns1.41414141.club`.
5. The victim Windows DNS Server processes and caches this response.

6. The next time we query for a subdomain of `deadbeef.fun`, the target Windows DNS Server will also query `ns1.41414141.club` for its response, as it is the NameServer for this domain.

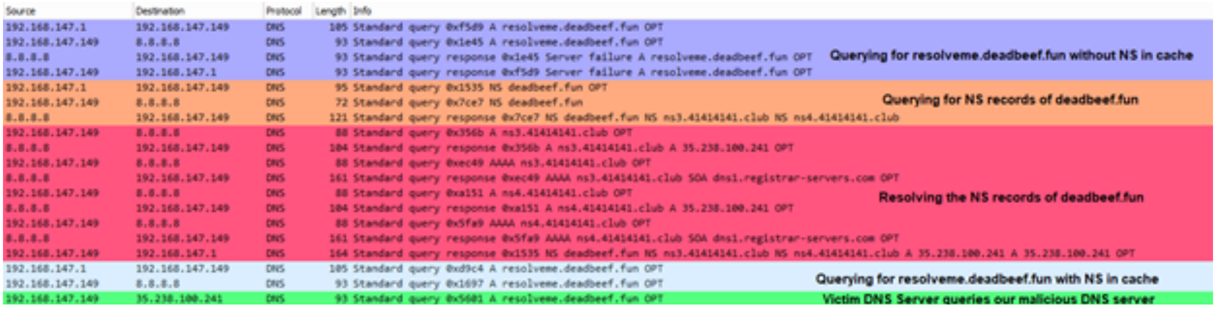

Figure 1: Packet capture of the victim DNS server querying our malicious server.

The Vulnerability – CVE-2020-1350

Function: `dns.exe!SigWireRead`
Vulnerability Type: Integer Overflow leading to Heap-Based Buffer Overflow

`dns.exe` implements a parsing function for every supported response type.

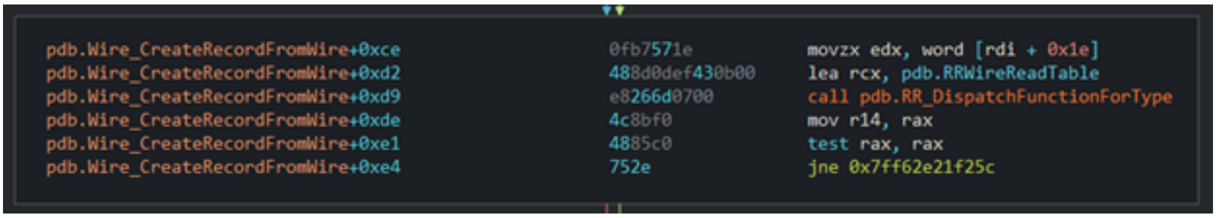

Figure 2: `Wire_CreateRecordFromWire`: `RRWireReadTable` is passed to `RR_DispatchFunctionForType` to determine the handling function.

```
;-- pdb.RRWireReadTable:
pdb.RRWireReadTable            .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0x8        .qword 0x00000001400ae330 ; pdb.AWireRead
pdb.RRWireReadTable+0x10       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x18       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x20       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x28       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x30       .qword 0x00000001400ae510 ; pdb.SoaWireRead
pdb.RRWireReadTable+0x38       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x40       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x48       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x50       .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0x58       .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0x60       .qword 0x00000001400ae3d0 ; pdb.PtrWireRead
pdb.RRWireReadTable+0x68       .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0x70       .qword 0x00000001400ae740 ; pdb.MinfoWireRead
pdb.RRWireReadTable+0x78       .qword 0x00000001400ae460 ; pdb.MxWireRead
pdb.RRWireReadTable+0x80       .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0x88       .qword 0x00000001400ae740 ; pdb.MinfoWireRead
pdb.RRWireReadTable+0x90       .qword 0x00000001400ae460 ; pdb.MxWireRead
pdb.RRWireReadTable+0x98       .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0xa0       .qword 0x00000001400ae370 ; pdb.CopyWireRead
pdb.RRWireReadTable+0xa8       .qword 0x00000001400ae460 ; pdb.MxWireRead
```

Figure 3: `RRWireReadTable` and some of its supported response types.

One of the supported response types is for a SIG query. According to Wikipedia, a SIG query is the "*signature record used in SIG(0) (RFC 2931) and TKEY (RFC 2930). RFC 3755 designated RRSIG as the replacement for SIG for use within DNSSEC.*"

Let's examine the disassembly generated by Cutter for `dns.exe!SigWireRead` – the handler function for the SIG response type:

```
pdb.SigWireRead+0x3b    8364242000    and   dword [var_148h], 0
pdb.SigWireRead+0x40    488d4c2430    lea   rcx, [var_138h]
pdb.SigWireRead+0x45    4c8bcf        mov   r9, rdi
pdb.SigWireRead+0x48    e87fdafcff    call  pdb.Name_PacketNameToCountNameEx
pdb.SigWireRead+0x4d    488be8        mov   rbp, rax
pdb.SigWireRead+0x50    4885c0        test  rax, rax
pdb.SigWireRead+0x53    74df          je    0x7ff62e21ed14

pdb.SigWireRead+0x55    0fb64c2430    movzx ecx, byte [var_138h]
pdb.SigWireRead+0x5a    482bf8        sub   rdi, rax
pdb.SigWireRead+0x5d    33d2          xor   edx, edx
pdb.SigWireRead+0x5f    6683c114      add   cx, 0x14 ; 20
pdb.SigWireRead+0x63    4533c0        xor   r8d, r8d
pdb.SigWireRead+0x66    6603cf        add   cx, di
pdb.SigWireRead+0x69    e85e710700    call  pdb.RR_AllocateEx
pdb.SigWireRead+0x6e    488bf0        mov   rsi, rax
pdb.SigWireRead+0x71    4885c0        test  rax, rax
pdb.SigWireRead+0x74    74be          je    0x7ff62e21ed14
```

Figure 4: Disassembly of `dns.exe!SigWireRead` as seen in Cutter.

The first parameter that is passed to `RR_AllocateEx` (the function responsible for allocating memory for the Resource Record) is calculated by the following formula:

[`Name_PacketNameToCountNameEx` result] + [0x14] + [The Signature field's length (`rdi`–`rax`)]

The signature field size may vary as it is the primary payload of the SIG response.

```
RFC 2535                DNS Security Extensions               March 1999

                        1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |        type covered           |  algorithm    |     labels    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                         original TTL                          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                      signature expiration                     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                      signature inception                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            key  tag           |                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+         signer's name         +
   |                                                               /
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-/
   /                                                               /
   /                            signature                          /
   /                                                               /
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5: The structure of SIG Resource Record according to RFC 2535.

As you can see in the image below, `RR_AllocateEx` expects its parameters to be passed in **16bit registers** as it only uses the `dx` part of `rdx` and `cx` part of `rcx`.

This means that if we can make the above formula output a result bigger than 65,535 bytes (the maximum value for a 16 bit integer), we have an integer overflow that leads to a much smaller allocation than expected, which hopefully leads to a heap based buffer overwrite.

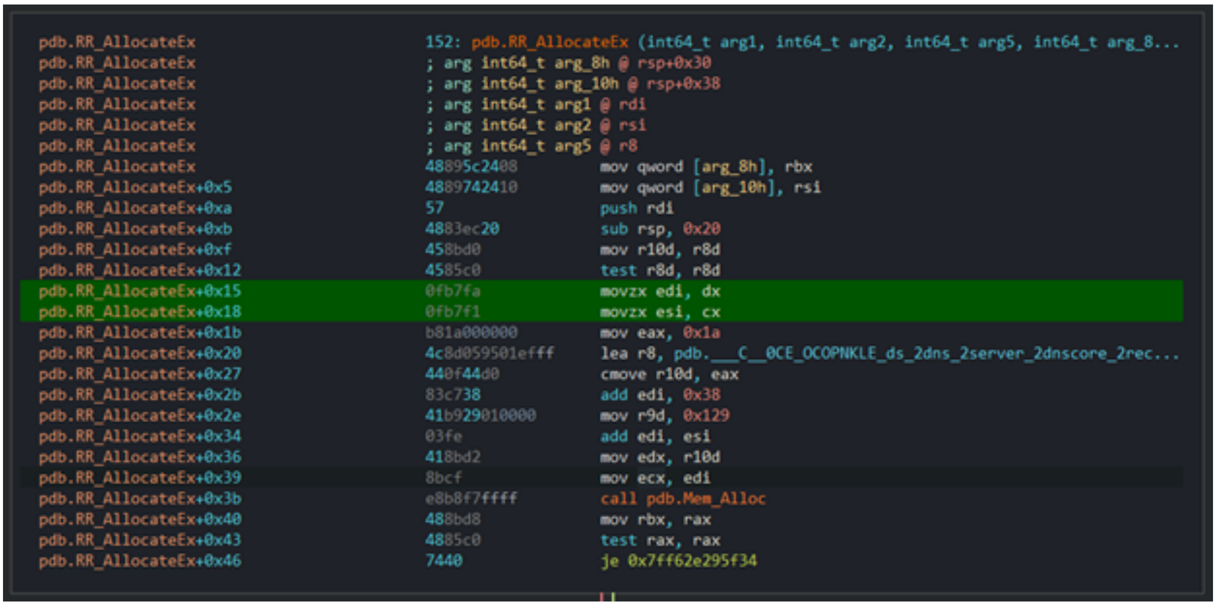

Figure 6: `RR_AllocateEx` converts its parameters to their 16bit value.

Conveniently enough, this allocated memory address is then passed as a destination buffer for `memcpy`, leading to a Heap-Based buffer overflow.

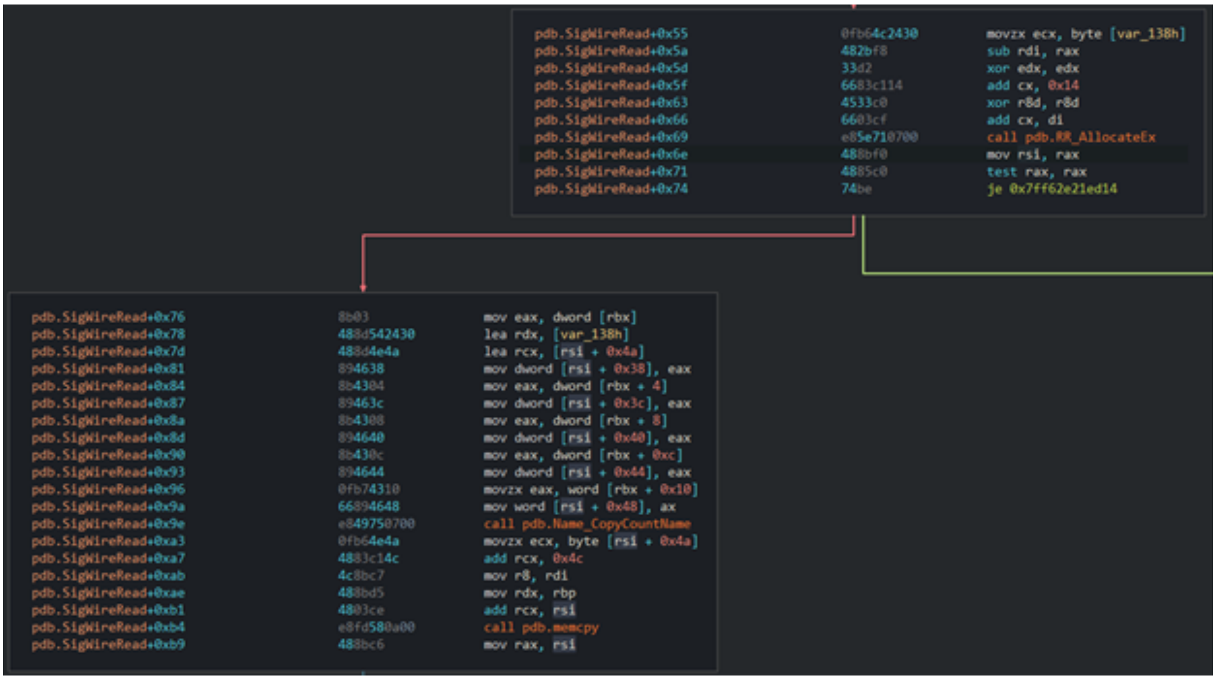

Figure 7: The allocated buffer from `RR_AllocateEx` is passed into `memcpy`.

To summarize, by sending a DNS response that contains a large (bigger than 64KB) SIG record, we can cause a controlled heap-based buffer overflow of roughly 64KB over a small allocated buffer.

Triggering the Vulnerability

Now that we're able to get the victim DNS server to query our DNS server for various questions, we have effectively turned it into a client. We can make the victim DNS server ask our malicious DNS server specific types of queries, and respectively answer with matching malicious responses.

We thought that all we needed to trigger this vulnerability was to make the victim DNS server query us for a SIG record, and answer it a SIG response with a lengthy signature (length >= 64KB). We were disappointed to find that DNS over UDP has a size limit of 512 bytes (or 4,096 bytes if the server supports EDNS0). In any case, that is not enough to trigger the vulnerability.

But what happens if there's a legitimate reason for a server to send a response larger than 4,096 bytes? For example, a lengthy TXT response or a hostname that can be resolved to multiple IP addresses.

DNS Truncation – But Wait, There's More!

According to the DNS RFC 5966:
"In the absence of EDNS0 (Extension Mechanisms for DNS 0), the normal behavior of any DNS server needing to send a UDP response that would exceed the 512-byte limit is for the server to truncate the response so that it fits within that limit and then set the TC flag in the response header. When the client receives such a response, it takes the TC flag as an indication that it should retry over TCP instead."

Great! So we can set the `TC` (truncation) flag in our response, which causes the target Windows DNS Server to initiate a new TCP connection to our malicious NameServer, and we can pass a message larger than 4,096 bytes. But how much larger?

According to DNS RFC 7766:
"DNS clients and servers SHOULD pass the two-octet length field, and the message described by that length field, to the TCP layer at the same time (e.g., in a single "write" system call) to make it more likely that all the data will be transmitted in a single TCP segment."

As the first two bytes of the message represent its length, the maximum size of a message in DNS over TCP is represented as 16 bits and is therefore limited to 64KB.

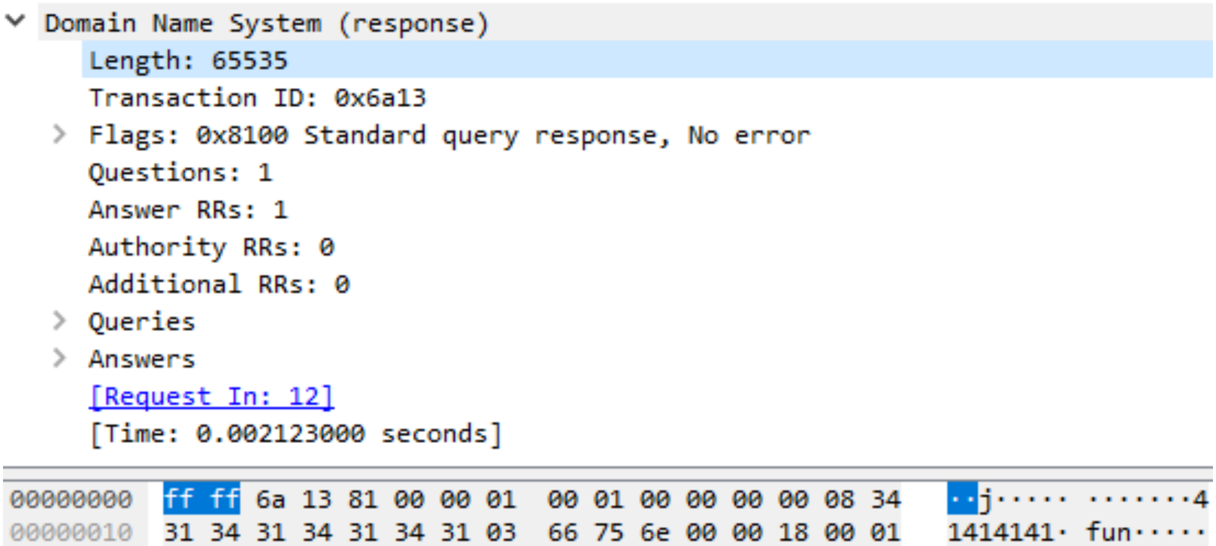

Figure 8: The first two bytes of a DNS over TCP message represent the message’s length.

But even a message of length 65,535 is not large enough to trigger the vulnerability, as the message length includes the headers and the original query. This overhead is not taken into consideration when calculating the size that is passed to `RR_AllocateEx`.

DNS Pointer Compression – Less is More

Let’s have another look at a legitimate DNS response (we chose a response of type A for convenience).

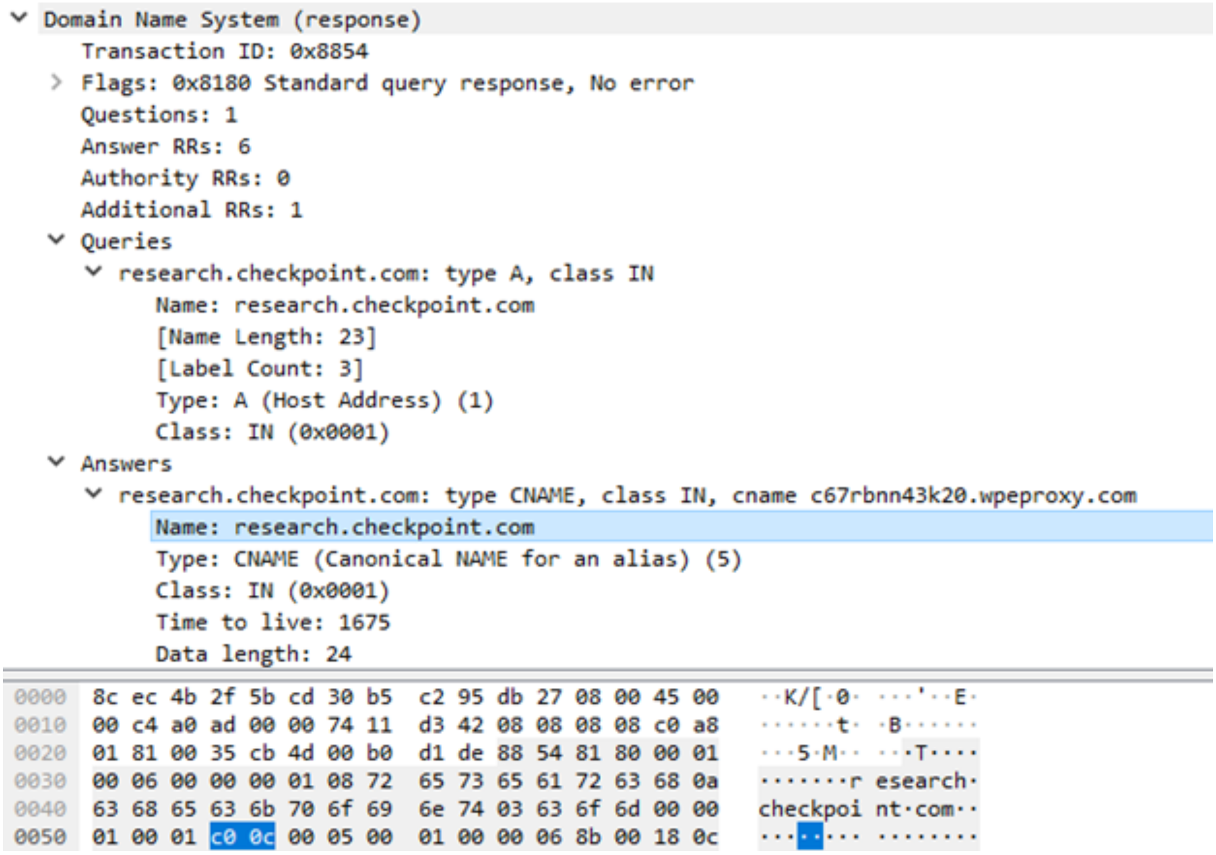

Figure 9: DNS response for `dig research.checkpoint.com A @8.8.8.8`, as seen in Wireshark.

You can see that Wireshark evaluated the bytes `0xc00c` in the answer's name field to `research.checkpoint.com`. The question is, why?

According to A warm welcome to DNS, powerdns.org:
"To squeeze as much information as possible into the 512 bytes, DNS names can (and often MUST) be compressed… In this case, the DNS name of the answer is encoded as 0xc0 0x0c. The c0 part has the two most significant bits set, indicating that the following 6+8 bits are a pointer to somewhere earlier in the message. In this case, this points to position 12 (= 0x0c) within the packet, which is immediately after the DNS header."

What is at the offset 0x0c (12) from the beginning of the packet?
It's `research.checkpoint.com`!

In this form of compression, the pointer points at the start of an encoded string. In DNS, strings are encoded as a (<size><value>) chain.

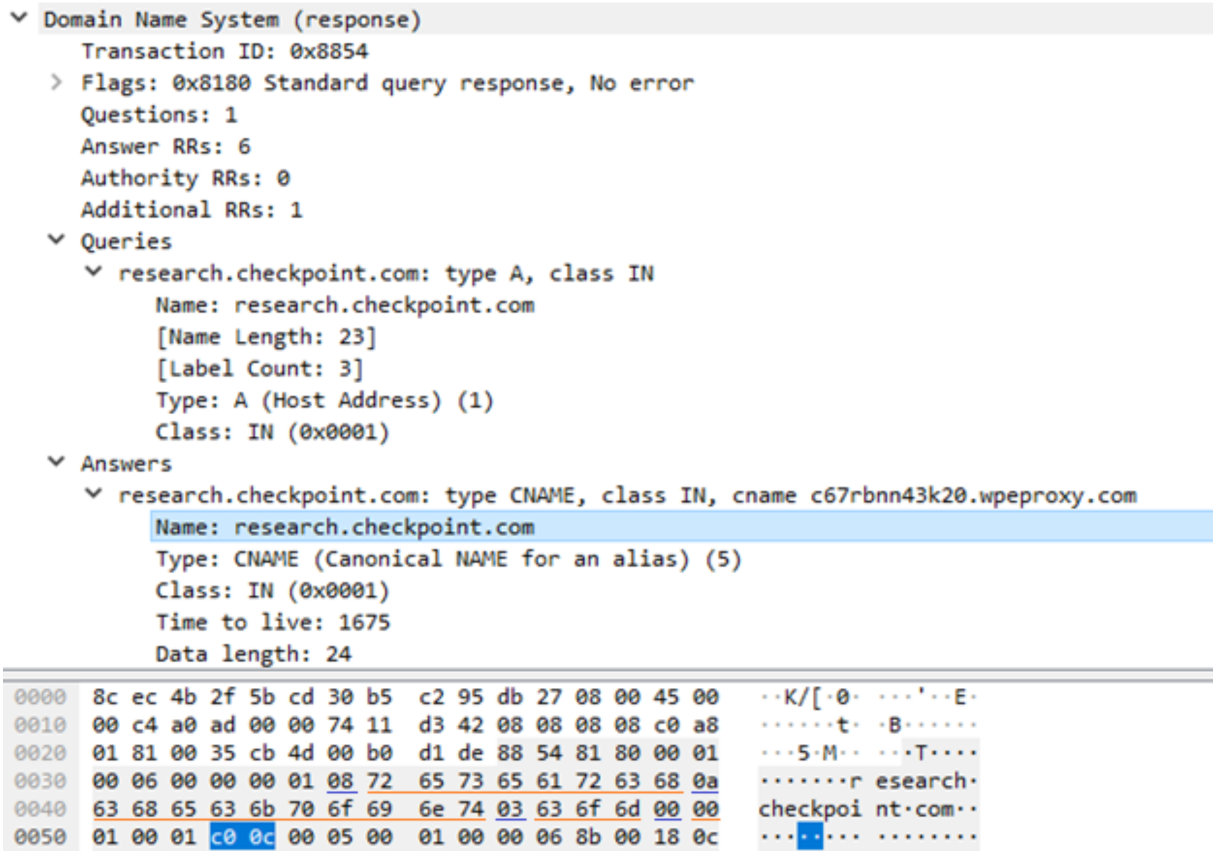

Figure 10: An illustration of a <size><value> chain.

So we can use the “magic” byte `0xc0` to reference strings from within the packet. Let’s once again examine the formula that calculates the size that is passed to `RR_AllocateEx`:

[`Name_PacketNameToCountNameEx` result] + [0x14] + [The Signature field’s length (`rdi`—`rax`)]

Reversing `Name_PacketNameToCountNameEx` confirms the behavior we described above. The purpose of `Name_PacketNameToCountNameEx` is to calculate the size of a name field, taking pointer compression into consideration. Having a primitive that allows us to increase the size of the allocation by a large amount, when only representing it with two bytes, is exactly what we need.

Therefore, we can use the pointer compression in the SIG Signer’s Name field. However, simply specifying `0xc00c` as the Signer’s name would not cause the overflow, as the queried domain name is already present in the query, and the overhead size is subtracted from the allocated value. But what about `0xc00d`? The only constraint we have to satisfy is that our encoded string is valid (ending with `0x0000`), and we can do it easily because we have a field without any character constraints – the signature value. For the domain `41414141.fun`, `0xc00d` points at the first character of the domain ('4'). The ordinal value of this character is then used as the size of the uncompressed string ('4' represents the value 0x34 (52)). Aggregation of the size of this uncompressed string, with the maximum amount of data we can fit in the Signature field (up to 65,535, depending on the original query), results in a value greater than 65,535 bytes, thus causing the overflow!

Let's test this with WinDBG attached to `dns.exe`:

```
(ec8.c24): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
msvcrt!memmove+0xaf:
00007ffb`cb19175e 660f7f41e0      movdqa  xmmword ptr [rcx-20h],xmm0 ds:0000003d`e2311000=????????????????????????????????
0:002> kv
Child-SP          RetAddr           : Args to Child                                                           : Call Site
0000003d`d721f358 00007ff6`fdeeed99 : 0000003d`eb00ee92 0000003d`eb00ee92 0000003d`eb00ee92 00000000`00000800 : msvcrt!memmove+0xaf
0000003d`d721f360 00007ff6`fdeef2a6 : 00000000`0000ffd5 0000003d`d721f630 00000000`00000000 00007ff6`fdeef4a7 : dns!SigWireRead+0xb9
0000003d`d721f4d0 00007ff6`fdedc12e : 0000003d`eb00e060 0000003d`d721f650 00000000`00000001 0000003d`eb00ee75 : dns!Wire_CreateRecordFromWire+0x15e
0000003d`d721f550 00007ff6`fdecb8a7 : 0000003d`eb00e060 00000000`00100461 00000000`00000000 0000003d`d7210018 : dns!Recurse_CacheMessageResourceRecords+0xfba
0000003d`d721f710 00007ff6`fde672a1 : 0000003d`eb00e060 0000003d`d721f7a0 00000000`00040000 00000000`00000004 : dns!Recurse_ProcessResponse+0x5ab
0000003d`d721f770 00007ff6`fdeffab0 : 0000003d`eb00e060 00000000`0000a19c 0000003d`d721f848 00000000`00002000 : dns!Answer_ProcessMessage+0x3d1
0000003d`d721f7e0 00007ff6`fde8304f : 00000000`0000000f 00000000`00012832 00000000`00000001 00000000`0000000f : dns!Tcp_Receiver+0x71c
0000003d`d721f890 00007ff6`fde819f7 : 00007ff6`00000000 0000003d`d2f75f00 00007ff6`fde45740 00007ff6`fdfdb980 : dns!loadDatabaseAndRunDns+0xd03
0000003d`d721f910 00007ffb`cb135ada : 00007ffb`cb135aa0 00000000`00000000 0000003d`d3024fd0 00000000`00000000 : dns!startDnsServer+0x467
0000003d`d721f9b0 00007ffb`cab013d2 : 00007ffb`cb135aa0 0000003d`d3024fd0 00000000`00000000 00000000`00000000 : sechost!ScSvcCtrlThreadW+0x3a
0000003d`d721f9e0 00007ffb`cba954f4 : 00007ffb`cab013b0 00000000`00000000 00000000`00000000 00000000`00000000 : KERNEL32!BaseThreadInitThunk+0x22
0000003d`d721fa10 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!RtlUserThreadStart+0x34
```

We crashed!

Although it seems that we crashed because we were trying to write values to unmapped memory, the heap can be shaped in a way that allows us to overwrite some meaningful values.

Previous exploitation attempts for `dns.exe` are available online. For example: A deeper look at ms11-058.

Triggering From the Browser

We know this bug can be triggered by a malicious actor who is present in the LAN environment. However, we thought it would be interesting to see if this bug can be triggered remotely without LAN access.

Smuggling DNS inside HTTP

By now you should be aware that DNS can be transported over TCP and that Windows DNS Server supports this connection type. You should also be familiar with the structure of DNS over TCP, but just in case, here's a quick review:

DNS/TCP
Length (16bit)
Transaction ID (16bit)
Flags (16bit)
Questions (16bit)
Answer RRs (16bit)
Authority RRs (16bit)
Additional RRs (16bit)
Queries
Answers
Authority Records
Additional Records

Figure 11: DNS over TCP message format.

Consider the following standard HTTP payload:

```
0000   50 4f 53 54 20 2f 70 77 6e 20 48 54 54 50 2f 31   POST /pwn 
HTTP/1
0010   2e 31 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d   .1..Accept: 
*/*.
0020   0a 52 65 66 65 72 65 72 3a 20 68 74 74 70 3a 2f   .Referer: 
http:/
```

Even though this is an HTTP payload, sending it to our target DNS server on port 53 causes the Windows DNS Server to interpret this payload as if it was a DNS query. It does this using the following structure:

```
0000   50 4f 53 54 20 2f 70 77 6e 20 48 54 54 50 2f 31   POST /pwn 
HTTP/1
0010   2e 31 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d   .1..Accept: 
*/*.
0020   0a 52 65 66 65 72 65 72 3a 20 68 74 74 70 3a 2f   .Referer: 
http:/
```

```
Message Length: 20559 (0x504f)
Transaction ID: 0x5354
Flags: 0x202f
Questions: 28791 (0x7077)
Answer RRs: 28192 (0x6e20)
Authority RRs: 18516 (0x4854)
Additional RRs: 21584 (0x5450)
Queries: [...]
```

Fortunately, Windows DNS Server supports both "Connection Reuse" and "Pipelining" of RFC 7766, which means we can issue multiple queries over a single TCP session and we can do so without waiting for replies.

Why is this important?

We can use basic JavaScript to issue a POST request to the DNS Server from the browser when a victim visits a website we control. But as shown above, the POST request is interpreted in a manner we don't really control.

However, we can abuse the "Connection Reuse" and "Pipelining" features by sending an HTTP POST request to the target DNS server (`https://target-dns:53/`) with binary data, containing another "smuggled" DNS query in the POST data, to be queried separately.

Our HTTP payload consists of the following:

- HTTP request headers that we do not control (`User-Agent`, `Referer`, etc).
- "Padding" so that the first DNS query has a proper length (0x504f) inside the POST data.
- Our "smuggled" DNS query inside the POST data.

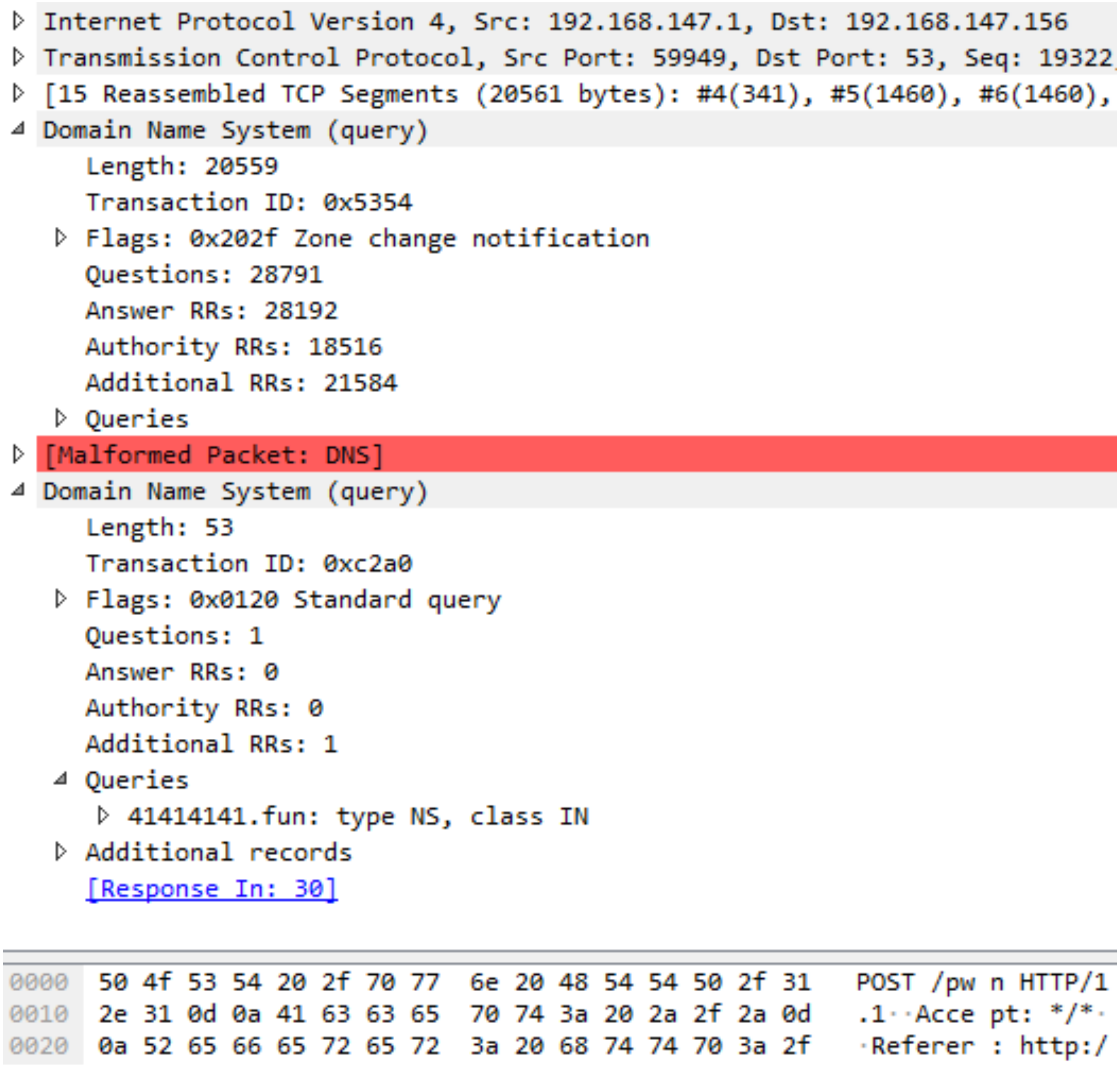

Figure 12: Multiple queries over a single TCP session as seen in Wireshark.

In practice, most popular browsers (such as Google Chrome and Mozilla Firefox) do not allow HTTP requests to port 53, so this bug can only be exploited in a limited set of web browsers – including Internet Explorer and Microsoft Edge (non-Chromium based).

Variant Analysis

The primary reason that this bug exists is because the `RR_AllocateEx` API expects a size parameter of 16 bits. It is generally safe to assume that the size of a single DNS message does not exceed 64KB and thus this behavior should not present an issue. However, as we just saw, this assumption is wrong when the result of `Name_PacketNameToCountNameEx` is taken into consideration while calculating the size of the buffer. This happens because the `Name_PacketNameToCountNameEx` function calculates the effective size of the uncompressed name and not the number of bytes it took to represent it in the packet.

To find other variants of this bug, we need to find a function that satisfies the following conditions:

- `RR_AllocateEx` is called with a variable size (and not a constant value).
- There is a call to `Name_PacketNameToCountNameEx` and its result is used to calculate the size passed to `RR_AllocateEx`.
- The value that is passed to `RR_AllocateEx` is calculated using values in the range of 16bits or more.

The only other function in `dns.exe` that satisfied these three conditions is `NsecWireRead`. Let's examine the following simplified code snippet we deduced from decompiling the function:

```
RESOURCE_RECORD* NsecWireRead(PARSED_WIRE_RECORD *pParsedWireRecord, DNS_PACKET *pPacket, BYTE
*pRecordData, WORD wRecordDataLength)
{
DNS_RESOURCE_RECORD *pResourceRecord;
unsigned BYTE *pCurrentPos;
unsigned int dwRemainingDataLength;
unsigned int dwBytesRead;
unsigned int dwAllocationSize;
DNS_COUNT_NAME countName;
pResourceRecord = NULL;
pCurrentPos = Name_PacketNameToCountNameEx(&countName, pPacket, pRecordData, pRecordData +
wRecordDataLength, 0);
if (pCurrentPos)
{
if
(pCurrentPos >= pRecordData // <-- Check #1 - Bounds check
&& pCurrentPos - pRecordData <= 0xFFFFFFFF // <-- Check #2 - Same bounds check (?)
&& wRecordDataLength >= (unsigned int)(pCurrentPos - pRecordData)) // <-- Check #3 - Bounds check
{
dwRemainingDataLength = wRecordDataLength - (pCurrentPos - pRecordData);
dwBytesRead = countName.bNameLength + 2;
// size := len(countName) + 2 + len(payload)
dwAllocationSize = dwBytesRead + dwRemainingDataLength;
if (dwBytesRead + dwRemainingDataLength >= dwBytesRead // <-- Check #4 - Integer Overflow check (32 bits)
&& dwAllocationSize <= 0xFFFF) // <-- Check #5 - Integer Overflow check (16 bits)
{
pResourceRecord = RR_AllocateEx(dwAllocationSize, 0, 0);
if (pResourceRecord)
{
Name_CopyCountName(&pResourceRecord->data, &countName);
memcpy(&pResourceRecord->data + pResourceRecord->data->bOffset + 2, pCurrentPos, dwRemainingDataLength);
}
}
}
}
}
```

```
return pResourceRecord;
}
```

As you can see, this function contains many security checks. One of them (Check #5) is a 16 bit overflow check that prevents the variant of our vulnerability in this function. We would also like to mention that this function has many more security checks than the average function in `dns.exe`, which makes us wonder if this bug was already noticed and fixed, but only in that specific function.

As we mentioned previously, Microsoft implemented the DNS client and DNS server in two different modules. While our vulnerability definitely exists in the DNS server, we wanted to see if it exists in the DNS client as well.

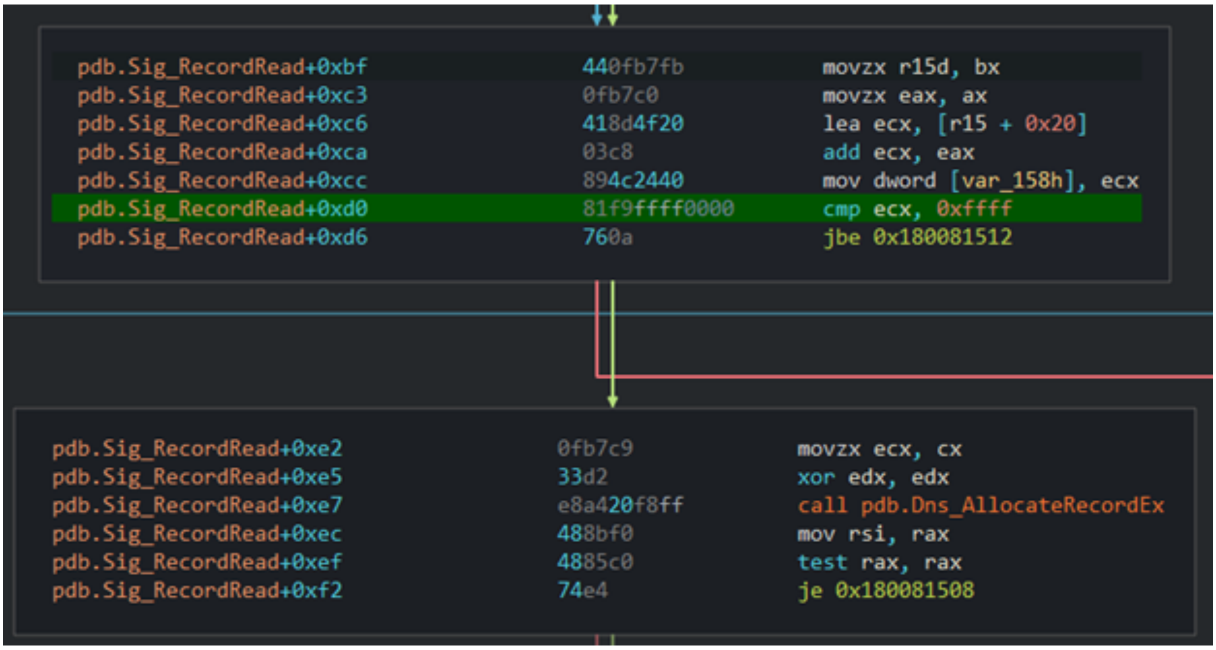

Figure 13: Disassembly snippet of `Sig_RecordRead` from `dnsapi.dll`.

It appears that, unlike `dns.exe!SigWireRead`, `dnsapi.dll!Sig_RecordRead` *does* validate at `Sig_RecordRead+D0` that the value that is passed to `dnsapi.dll!Dns_AllocateRecordEx` is less than 0xFFFF bytes, thus preventing the overflow.

The fact that this vulnerability does not exist in `dnsapi.dll`, as well as having different naming conventions between the two modules, leads us to believe that Microsoft manages two completely different code bases for the DNS server and the DNS client, and does not synchronize bug patches between them.

Exploitation Plan

Per Microsoft's request, we decided to withhold information about the exploitation primitives in order to give users enough time to patch their DNS servers. Instead, we discuss our exploitation plan as it applies to Windows Server 2012R2. However, we do believe that this plan should apply to other versions of Windows Server as well.

The `dns.exe` binary was compiled with Control Flow Guard (CFG), which means the traditional approach of overwriting a function pointer in memory is not enough to exploit this bug. If this binary was not compiled with CFG, exploiting this bug would be pretty straight-forward, as quite early on we encountered the following crash:

```
0:003> r
rax=0020a0a0a0a0a0a0 rbx=0000001cd935aa80 rcx=4141414141414141
rdx=00007df5ffa60000 rsi=4141414141414141 rdi=4141414141414141
rip=00007ffbe3e9166e rsp=0000001cd8a6fa98 rbp=0000000000000001
 r8=0000001cd8a6fb20  r9=0000001cdd9c8040 r10=0000000000000000
r11=0000000000000287 r12=0000001cd8a6fb70 r13=00007ff7e4671338
r14=0000001cd935aa80 r15=0000001cd8a6fb20
iopl=0         nv up ei pl nz na po cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010207
ntdll!LdrpValidateUserCallTarget+0xe:
00007ffb`e3e9166e 488b14c2        mov     rdx,qword ptr [rdx+rax*8] ds:010582fb`04ab0500=????????????????
0:003> k
Child-SP          RetAddr           Call Site
0000001c`d8a6fa98 00007ffb`e3e6e0a6 ntdll!LdrpValidateUserCallTarget+0xe
0000001c`d8a6faa0 00007ffb`e3e6de83 ntdll!FindNodeOrParent+0x42
0000001c`d8a6fae0 00007ff7`e46f5775 ntdll!RtlDeleteElementGenericTableAvl+0x13
0000001c`d8a6fb10 00007ff7`e46ecc57 dns!DnsRq_FindExpiredQuery+0x159
0000001c`d8a6fb70 00007ff7`e47205bf dns!Recurse_RecursionTimeoutThread+0x377
0000001c`d8a6fbd0 00007ffb`e3a71412 dns!threadTopFunction+0x7f
0000001c`d8a6fc10 00007ffb`e3e254f4 KERNEL32!BaseThreadInitThunk+0x22
0000001c`d8a6fc40 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

Figure 14: Crash at `ntdll!LdrpValidateUserCallTarget`.

As you can see, we crashed at `ntdll!LdrpValidateUserCallTarget`. This is the function responsible for validating function pointer targets as part of CFG. We can see that the pointer to be validated (`rcx`) is fully controllable, which means that we successfully overwrote a function pointer somewhere along the way. The reason we saw a crash is that the function pointer is used as an index to a global bitmap table with "allowed" / "disallowed" bit per address, and our arbitrary address led to a read from an unmapped page in the table itself.

To exploit this bug to a full Remote Code Execution while defeating CFG, we need to find primitives that give us the following capabilities: write-what-where (to precisely overwrite a return address on the stack) and an infoleak (to leak memory addresses, such as the stack).

Infoleak

In order to achieve an Infoleak primitive, we corrupted the metadata of a DNS resource record, while it is still in the cache, using our overflow. Then, when queried again from the cache, we were able to leak adjacent heap memory.

WinDNS’ Heap Manager

WinDNS uses the function `Mem_Alloc` to dynamically allocate memory. This function manages its own memory pools to be used as an efficient cache. There are 4 memory pool buckets for different allocation sizes (up to 0x50, 0x68, 0x88, 0xA0). If the requested allocation size is greater than 0xA0 bytes, it defaults to `HeapAlloc`, which uses the native Windows heap. The heap manager allocates an additional 0x10 bytes for the memory pool header, which contains metadata including the buffer’s type (allocated / free), a pointer to the next available chunk of memory, a cookie for debug checks, and more. The heap manager implemented its allocation lists in a singly-linked-list fashion, meaning that chunks are allocated in the reverse order that they were freed (LIFO).

Write-What-Where

To achieve a write-what-where primitive, we attacked the WinDNS heap manager by corrupting a chunk’s header (metadata), de-facto corrupting the freelist.

After the freelist is corrupted, the next time we try to allocate anything of the right size, the memory allocator assigns a memory region of our choice for us as a writable allocation – a “Malloc-Where” exploit primitive.

To bypass CFG, we want that memory region to be on the stack (whose location we hopefully know thanks to the infoleak). Once we have a write capability on the stack, we can overwrite a return address to an address we want to execute, effectively hijacking the execution flow.

It is important to mention that by default, the DNS service restarts in the first 3 crashes, increasing the chances for successful exploitation.

Conclusion

This high-severity vulnerability was acknowledged by Microsoft and was assigned CVE-2020-1350.

We believe that the likelihood of this vulnerability being exploited is high, as we internally found all of the primitives required to exploit this bug. Due to time constraints, we did not continue to pursue the exploitation of the bug (which includes chaining together all of the exploitation primitives), but we do believe that a determined attacker will be able to exploit it. Successful exploitation of this vulnerability would have a severe impact, as you can often find unpatched Windows Domain environments, especially Domain Controllers. In addition, some Internet Service Providers (ISPs) may even have set up their public DNS servers as WinDNS.

We strongly recommend users to patch their affected Windows DNS Servers in order to prevent the exploitation of this vulnerability.

As a temporary workaround, until the patch is applied, we suggest setting the maximum length of a DNS message (over TCP) to 0xFF00, which should eliminate the vulnerability. You can do so by executing the following commands:

```
reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DNS\Parameters" /v
"TcpReceivePacketSize" /t REG_DWORD /d 0xFF00 /f
net stop DNS && net start DNS
```

Check Point IPS blade provides protection against this threat: "Microsoft Windows DNS Server Remote Code Execution (CVE-2020-1350)"

Check Point SandBlast Agent E83.11 already protects against this threat

Disclosure Timeline

- 19 May 2020 – Initial report to Microsoft.
- 18 Jun 2020 – Microsoft issued CVE-2020-1350 to this vulnerability.
- 09 Jul 2020 – Microsoft acknowledged this issue as a wormable, critical vulnerability with a CVSS score of 10.0.
- 14 Jul 2020 – Microsoft released a fix (Patch Tuesday).