Statement from NDSS 2025: The NDSS 2025 PC appreciated the technical contributions made in this paper, the confirmation of prior work that is otherwise not (directly) reproducible, and the contributions towards fostering future anti-censorship research, but it also found the paper highly controversial because the experiments that the authors conducted raise ethical concerns. This paper went through scrutiny by various stakeholders beyond the regular PC review, including evaluation by the NDSS'25 Ethics Review Board and consultation of the Steering Committee and ISOC. While the ethical ambiguities were deemed remedied after data aggregation and deletion, the IRB Exempt decision that the authors received from their institution should have been questioned and repudiated by the authors as there are clear human risks involved. Questioning such an IRB decision should be an obligation by researchers in the security community. Additionally, the PC does not consider itself qualified to make a judgment about the legal implications of this work. We acknowledge that there were conflicting opinions during the broader review process on whether the benefits of this research outweigh its risks. We hope that the acceptance of this paper helps the community understand the possible impact of research work, allows better mechanisms to deal with similar cases, and contributes to developing accepted standards on when and how such types of offensive research can be done. The acceptance of this paper does not constitute the PC's endorsement of the used methodology. We advise authors to seek legal advice (from different legislations if applicable) before/while doing security research that may impact critical targets.

Wallbleed: A Memory Disclosure Vulnerability in the Great Firewall of China

Shencha Fan
GFW Report
gfw.report@protonmail.com

Jackson Sippe
University of Colorado Boulder
Jackson.Sippe@colorado.edu

Sakamoto San
Shinonome Lab
54k4m070@proton.me

Jade Sheffey
University of Massachusetts Amherst
jsheffey@cs.umass.edu

David Fifield
david@bamsoftware.com

Amir Houmansadr
University of Massachusetts Amherst
amir@cs.umass.edu

Elson Wedwards
ElsonWedwards@proton.me

Eric Wustrow
University of Colorado Boulder
ewust@colorado.edu

***Abstract*—We present *Wallbleed*, a buffer over-read vulnerability that existed in the DNS injection subsystem of the Great Firewall of China. Wallbleed caused certain nation-wide censorship middleboxes to reveal up to 125 bytes of their memory when censoring a crafted DNS query. It afforded a rare insight into one of the Great Firewall's well-known network attacks, namely DNS injection, in terms of its internal architecture and the censor's operational behaviors.**

To understand the causes and implications of Wallbleed, we conducted longitudinal and Internet-wide measurements for over two years from October 2021. We (1) reverse-engineered the injector's parsing logic, (2) evaluated what information was leaked and how Internet users inside and outside of China were affected, and (3) monitored the censor's patching behaviors over time. We identified possible internal traffic of the censorship system, analyzed its memory management and load-balancing mechanisms, and observed process-level changes in an injector node. We employed a new side channel to distinguish the injector's multiple processes to assist our analysis. Our monitoring revealed that the censor coordinated an incorrect patch for Wallbleed in November 2023 and fully patched it in March 2024.

Wallbleed exemplifies that the harm censorship middleboxes impose on Internet users is even beyond their obvious infringement of freedom of expression. When implemented poorly, it also imposes severe privacy and confidentiality risks to Internet users.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230237
www.ndss-symposium.org

I. Introduction

The national Internet censorship system in China, known as the Great Firewall (GFW), is composed of many parts and subsystems, each using different techniques to control access to online information. One prominent component is the *DNS injection* subsystem, which forges DNS responses to DNS queries for censored domain names. Until March 2024, certain DNS injection devices had a parsing bug that would, under certain conditions, cause them to include up to 125 bytes of their own memory in the forged DNS responses they sent. We call this bug Wallbleed, as a nod to similar buffer over-read vulnerabilities like Heartbleed [1], Ticketbleed [2], and Cloudbleed [3], [4].

In this work,[1] we analyze the causes and implications of Wallbleed. Our study confirms that Wallbleed existed for at least two years. (Reports of a similar vulnerability circulated as early as 2010 [5], [6].) We ran continuous measurements

[1]Project homepage: https://gfw.report/publications/ndss25/en/.

between October 2021 and April 2024. The vulnerability was partially patched in November 2023, but DNS injectors were still vulnerable to certain crafted queries until March 2024, when it was finally fully patched.

Wallbleed provides an unprecedented look at the GFW, both its internal architecture as well as the censor's operational behaviors. While prior work has studied what domains and resources are blocked in China, little is known about the inner workings of the GFW's network middleboxes [7]–[9]. From the data leaked with Wallbleed, we are able to discern the underlying architecture of the GFW, and we reverse-engineer the parsing bug responsible for Wallbleed to create a behavior-identical implementation in C. During the course of our study, we discovered previously unknown characteristics of GFW DNS injection, such as that every injection process cycles through a list of false IP addresses independently and in a fixed order, a side channel that distinguishes multiple processes in an injector node. Finally, we conducted longitudinal and Internet-wide measurements to monitor the censor's patching activity, taking advantage of a rare opportunity to learn more about how the GFW maintains its censorship infrastructure.

We examined the contents of Wallbleed-leaked memory and discovered apparent network protocol headers, payloads, x86_64 stack frames, and executable code (though we show evidence that it is not the code of the GFW itself). We sent traffic tagged with recognizable byte patterns past the GFW, and in some cases recovered those tags in subsequent Wallbleed responses, demonstrating that the vulnerability leaked at least some of the traffic seen by the GFW. We see, in Wallbleed-leaked memory, samples of plaintext network traffic and protocols not all of which are related to DNS censorship, including IP, TCP, UDP, and HTTP. We also performed IPv4-wide scans to estimate how many addresses inside and outside China might have had their traffic processed by Wallbleed-vulnerable middleboxes. Even some traffic whose source and destination are both outside China might have been affected, due to routing through China's network border.

Significant ethical considerations accompany an investigation of this nature. We discuss them in depth in Section IX, including the question of whether or not to disclose a vulnerability in a system that is, itself, considered by many to be a source of harm [10], [11]. The injection of fraudulent DNS responses is one of many persistent attacks carried out daily by the GFW. The intention and effect of these attacks are well-known: to limit people's access to information. Wallbleed is an example of how censorship devices pose risks to security and privacy even beyond their obvious infringement of freedom of expression. While this specific vulnerability was eventually fixed, the existence of such devices continues to be a hazard.

II. Background

A. DNS injection attacks

The GFW's DNS injection subsystem employs a fleet of middlebox devices at China's network border that watch for DNS queries for blocked domain names. When they see one, they *inject* a DNS response back towards the client, spoofing the source address as if it came from the intended resolver. The injected response is a false answer to the query, containing an incorrect, useless IP address. When the client subsequently tries to connect, it will meet with an error rather than a connection to the expected destination. The injection middleboxes are on-path, not in-path devices: they do not block the query from reaching the legitimate resolver, nor the authentic response from reaching the client. The injectors' false response "wins" because it arrives first, having been injected at a point in the network path nearer to the client [12]. Clients accept the first received DNS response, as only one response is expected per query in normal operation.

The DNS injection subsystem is *bidirectional*: it responds to queries in both directions, whether they are leaving China or entering it. This feature is convenient for analysis: it is easier to send packets into China from the outside, than to acquire and maintain a network vantage point inside the country. By sending DNS queries to a non-live IP address in China, we can be sure that any responses received are from a middlebox injector, not the end host.

Just as the GFW consists of disparate components, DNS injection is done by several distinct kinds of DNS injector—at least three. The foundational pieces of research on this topic are gfwrev's work of 2009 [13], Anonymous's work of 2014 [7], Anonymous et al.'s "Triplet censors" of 2020 [8], and Hoang et al.'s "How great is the Great Firewall?" of 2021 [9]. The several kinds of injector differ in their lists of blocked domain names, their network fingerprints at the IP and DNS layers, and quirks of their parsing logic. The Wallbleed vulnerability exists in just one kind of injector, the one which Anonymous et al. call "Injector 3" [8 §4.1].

DNS injection has long been a primary technique of the GFW. But circumventing it alone is not enough, because there are other systems in play. Even if a client is able to get a correct DNS response by some means, their communication may be disrupted by a different subsystem, such as the IP address filter [14 §4.1], TLS SNI filter [14]–[16] or TLS ESNI filter [14], [17]. In this paper we will be concerned only with DNS injection, and with only one kind of DNS injector.

The government of China is not alone in using DNS injection for censorship. See also, for example, the "DNS Tampering" column of Table 1 of the 2023 survey of Master and Garman [18], and Nourin et al.'s study of bidirectional injection of DNS and other protocols in Turkmenistan [19].

B. The format of DNS messages

As the Wallbleed vulnerability stems from low-level parsing errors, it will be important to understand how DNS messages are represented on the wire and in memory. The format of DNS messages is specified in RFC 1035 [20]. Queries and responses have the same basic format: a 12-byte header, followed by four variable-length sections: *question*, *answer*, *authority*, and *additional*. We will be concerned only with the question and answer sections. The question section contains the DNS name being queried (or, in a response, the name that the response is in answer to). The answer section, present only in responses,

contains the information requested by a query (commonly an IP address) in a data structure called a *resource record*.

Figure 1 is a sample injected DNS response. It features everything that will be necessary to understand the DNS messages that arise in this research. We will use these field names and background colors consistently.

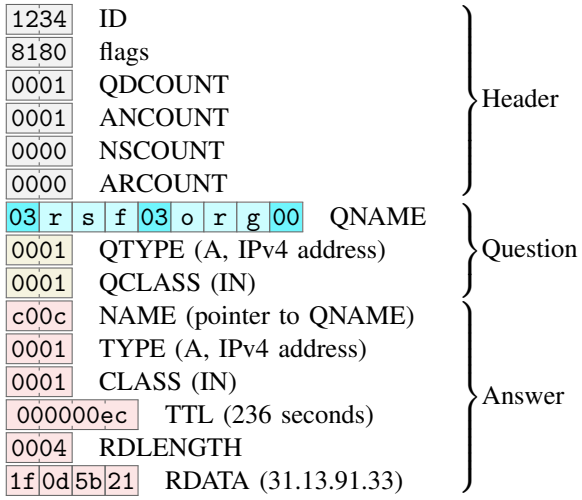

Fig. 1: The structure of an injected DNS response.

The message is a DNS response (rather than a query), as indicated by the most significant bit of the flags being set. It has one question and one answer; the authority and additional sections are empty. The QNAME in the question section is the name the DNS client asked to resolve, `rsf.org`. The answer section has been constructed by the GFW injector. It asserts that the client's QNAME resolves to an incorrect IPv4 address (one of hundreds the injector may use).

The most important thing to understand is the encoding of DNS *names*. Names are pervasive in the DNS protocol: there is one in every question section (QNAME field), and at least one in every resource record (NAME field). A name is a sequence of *labels*. A label, in turn, is sequence of bytes, prefixed by a byte indicating its length. A name ends at an empty label, i.e., one that consists only of the length prefix `00`. The name `example.com` has three labels of 7, 3, and 0 bytes. Its encoding is 13 bytes long: `07 e x a m p l e 03 c o m 00`.

There is an exception to the length-prefix encoding of names. If the two most significant bits of the the length prefix are set, then the other 6 bits of that byte, and the 8 bits of the next byte, form a 14-bit *compression pointer* that indicates that the remaining labels in the name are found starting at the given byte offset in the message. Message compression is useful because it is common for DNS messages to contain the same name more than once, or several names with a common suffix. The compression pointer pattern `c00c` is one to know by sight. It points to byte offset 12, which is the offset of the QNAME field in the question section. Rather than copy the QNAME into the answer section, the injectors vulnerable to Wallbleed begin the answer section with a `c00c` compression pointer. (The use of compression pointers is not unique to the GFW; legitimate resolvers use them as well. Of the various kinds of DNS injector that exist in the GFW, some use `c00c` and some make a copy of QNAME [8 §4.1].)

The format of DNS names, with its length prefixes and pointer indirection, lends itself to memory safety errors in parsers. When processing a label length prefix, one must check that the end of the label stays inside the bounds of the message. The lack of such a check is the fundamental cause of the Wallbleed overflow vulnerability.

III. Demonstrating Overflow

This is a well-formed query whose QNAME, `rsf.org`, is on the GFW's blocklist:

```
1234 0100 0001 0000 0000 0000 03 r s f 03
o r g 00 0001 0001
```

If we send these bytes, in a UDP datagram with destination port 53, from a host outside China to a host inside China, we get an injected DNS response. (Actually more than one response, because this QNAME is on the blocklist of more than one kind of injector.) Any destination IP address in China will do, even a non-responsive one—the query only needs to pass by an injector middlebox in transit.

An injected response looks like the following (this is Figure 1 in a more compact form):

```
1234 8180 0001 0001 0000 0000 03 r s f 03
o r g 00 0001 0001 c00c 0001 0001
000000ec 0004 1f 0d 5b 21
```

The ID and question section are copied from the query. The flags have been set as appropriate for a response. The answer section falsely asserts that the name `rsf.org` (represented by a compression pointer `c00c`) resolves to the IPv4 address 31.13.91.33 (`1f 0d 5b 21`). As detailed in Appendix A, this fake address is one of many that the injector may use. If we send the query again, we will likely get a different one.

See what happens if we now artificially increase the length prefix of the `org` label from `03` (3) to `20` (32):

```
1234 0100 0001 0000 0000 0000 03 r s f 20
o r g 00 0001 0001
```

For one thing, we now get only one injected response: the malformed query is ignored by injectors other than the Wallbleed ones. The TTL and IP address in the answer section are different than before, which is expected: these typically change in every response. More significantly, the injected response contains 29 additional bytes before the answer section. These bytes come from the memory of the injection device that handled the query. In this example, the leaked bytes are a fragment of a UPnP HTTP header:

```
1234 8180 0001 0001 0000 0000 03 r s f 20
o r g 00 0001 0001 C u s t o m / 1 .
0   U P n P / 1 . 0   P r o c / V
e r 0d c00c 0001 0001 00000082 0004 68 f4
2e a5
```

Whenever an injector responds to such a query, it reveals a small window of its memory, each time with different contents.

We posit that something like the following process must have occurred inside the injector device. Having observed a DNS query on its network tap, the injector copies the packet into memory for processing. Its goal is to parse the QNAME from the query, check it against a blocklist, and inject a response if needed. In parsing the QNAME, the injector first sees the 3-byte label `rsf`: so far, so good. But the length prefix `20` says that the next label is 32 bytes long, which extends over the `org` label and empty label, the QTYPE and QCLASS fields, and past the end of the query. Because it fails to enforce a bounds check, the injector regards the bytes that follow the packet in memory as being part of the query—as if the QNAME had been the 38 bytes `03 r s f 20 o r g 00 00 01` ⋯ `P r o c /`. Despite the extraneous bytes at the end of the name, it still matches the blocklist, for reasons we will explain in Section III-A. The injector copies the entire QNAME (as it sees it) into a DNS response. The next 4 bytes (in this example, `V e r 0d`) are interpreted as the query's QTYPE and QCLASS, and also copied into the response.

Why does the parser stop at the `/` byte, rather than treat it as a length prefix and reading another label? We present a precise, reverse-engineered description of the parsing algorithm in Appendix B, which answers this and other questions. In this case, it is because the QNAME parser stops after the first label length prefix that is past the end of the query.

A. *Blocklist matching*

When an injector checks a name against its blocklist, it does not use the name's wire-format representation. Instead, it flattens the QNAME into a dot-delimited string terminated by a `00` byte. This string is what is passed to the blocklist lookup function. The evidence for this claim is that when a label in a query contains an ASCII dot character `.` or a null byte `00`, the blocklist matcher interprets it as a label separator or name terminator, respectively. For example, if the name `example.com` were on the blocklist, either of the QNAMEs `07 e x a m p l e 03 c o m 00` or `0f e x a m p l e . c o m 00 a b c 00` would elicit an injection. Though the names are distinct at the DNS level (the first consists of three labels of 7, 3, and 0 bytes; the second of two labels of 15 and 0 bytes), they are both flattened into the same effective string "example.com".

This explains why the QNAME `03 r s f 20 o r g 00 00 01` ⋯ `P r o c /` that was understood by the injector in the previous example matches the blocklist rule for `rsf.org`. Though the second label is not just `org`, but rather `org` plus many additional bytes, the first of those additional bytes is `00`, which terminates the name when it is flattened into a string. The extra bytes are included in the injected DNS response, but they do not affect blocklist matching.

It also explains why we modified the length prefix of the `org` label, rather than the `rsf` label. If we had extended the `rsf` label, the `03` before `o r g` would have been interpreted as a literal character in the flattened name string—and because the string "rsf\x03org" does not match anything on the blocklist, it would not have gotten a response. Whereas by extending the `org` label, `rsf` and `org` remain separate labels, and the final empty label `00` becomes a string terminator. Altering the first length prefix can work, but then the second length prefix must also be changed to a dot, in order to separate the labels in the final string: `20 r s f . o r g 00`.

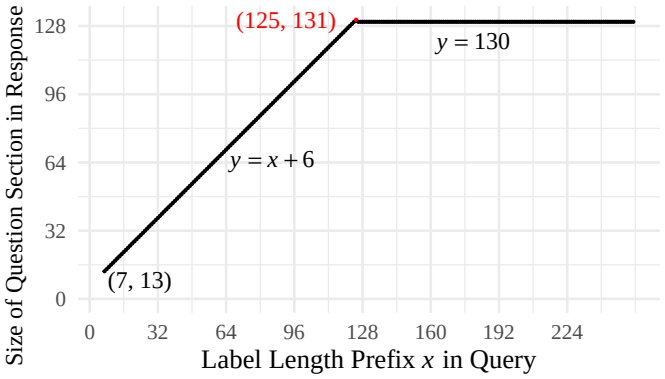

Fig. 2: The size of the question section in injected DNS responses versus the label length prefix x in a query for the QNAME x `r s f . o r g 00`. We are using the "embedded dot character" and "embedded null terminator" tricks from Section III-A, in order to place the variable label length prefix at the beginning of the question section.

Blocklist rules are not literal names, but *patterns*, like regular expressions [9 §4.1]. A single rule may, for example, block an entire domain with its subdomains. Patterns are not uniformly constructed (showing signs of fallible human curation). The `rsf.org` pattern we have been using is end-anchored and label-anchored: `rsf.org` and `x.rsf.org` match the pattern, but `xrsf.org`, `rsf.org.x`, and `rsf.orgx` do not. As a regular expression, it would be something like `(.*\.)*rsf\.org$`. In comparison, the pattern for `shadowvpn.com` is start-anchored and not label-anchored: `shadowvpn.com`, `shadowvpn.comx`, and `shadowvpn.com.x` match it, but `xshadowvpn.com` and `x.shadowvpn.com` do not. Its regular expression would be `^shadowvpn\.com.*`.

B. *Maximizing leaked bytes per response*

The greatest number of bytes that may be leaked in a single response is 125. This is a consequence of the fact that the question section in injected responses has a maximum size of 131 bytes, and the shortest question section in a query that triggers a response has a length of 6 bytes. The question section in a response contains a copy of the question section from the query at the beginning; everything after that is leaked memory. To maximize the amount of leaked memory, minimize the size of the question section in the query (how big the query actually is), and maximize the size of the question section in the response (how big the injector *thinks* the query is).

The first step in minimizing the size of the query is to omit the QTYPE and QCLASS fields. When these fields are absent,

the injector reads them from its own memory. QCLASS has no effect, and QTYPE only controls whether the injector crafts a type A (IPv4) or type AAAA (IPv6) response. The injectors default to type A for unknown QTYPEs; they send a type AAAA response only when the QTYPE is `001c`. In either case, the size of the question section is the same.

The other part of minimizing query size is to use a short QNAME. To find short DNS names that trigger an injection response, we enumerated all names of the forms $a.b$, $a.bc$, and $ab.c$, with a, b, and c ranging over the characters 'a'–'z', '0'–'9', '-', and '_', and sent them in DNS queries into China. We found eight short names that worked: `3.tt`, `4.tt`, `5.tt`, `6.tt`, `7.tt`, `8.tt`, `9.tt`, and `x.co`. [2] Each of these names takes 6 bytes to encode (e.g., `01 3 02 t t 00`). [3]

At the start of Section III, we caused an injector to leak 29 bytes by increasing a QNAME label length prefix from 3 to 32. Intuitively, in order to leak more bytes, one should increase the label length further. This intuition holds true, but only up to a point. Figure 2 shows how the size of the question section in a response varies as the label length in a query is increased. (The injector we are concerned with does not enforce RFC 1035's length limit of 63 bytes on labels [20 §4.1.4], naively interpreting every byte value as a simple length instead.) They increment one-for-one until the response question section reaches a maximum of 131 bytes. Beyond that point, the question section becomes slightly smaller than the maximum, 130 bytes.

This odd behavior is a product of the confused logic of the query parsing algorithm (Appendix B). Two conditions that cause the algorithm's main loop to terminate are when the total length of the QNAME exceeds 127 bytes while processing the contents of a label, and when the parser has just read a length prefix past the end of the query. The sweet spot of 131 bytes occurs when the QNAME is 127 bytes exactly (including the final label length prefix). In this case, the first exit condition is avoided, allowing the next iteration of the loop to read 1 additional byte before exiting. The 127 bytes of the QNAME, plus 4 bytes for the missing QTYPE and QCLASS, produce a question section of 131 bytes total.

The QNAME length limit is a general characteristic of this kind of injector, independent of the Wallbleed parsing bug. We sent well-formed queries for names of increasing length (`a.google.sm`, `aa.google.sm`, `aaa.google.sm`, ...) into China, using a base domain `google.sm` known to match the blocklist. The injectors stopped responding once the `m` byte at the end of the final label was pushed out of the first 127 bytes. The limit is the same for type A and type AAAA queries, and for any number of labels in the QNAME. The maximum name length prescribed in RFC 1035 is 255 bytes [20 §2.3.4].

Though it is satisfying to know the absolute limits and the reasons for them, there is little difference between 130 and 131 bytes in practice. In many of the experiments of this paper (some performed before we understood the nuances of the parsing algorithm), we used a label length prefix of `ff`, which gets 1 byte fewer per query than the maximum possible. A question section of 130 bytes in response to sufficiently large length prefixes agrees with findings of klzgrad in 2012 [6].

C. Incomplete patch (Wallbleed v2)

The GFW attempted to patch Wallbleed between September and November 2023, adding restrictions to the DNS message parsing algorithm. We have documented the progression of patching in Section VII. The QTYPE and QCLASS fields could no longer be omitted, and QCLASS had to have the value `0001`. In addition, a label length prefix that overflowed the end of the query but did not reach the 127-byte QNAME length threshold caused a query to be ignored. A query like the following no longer worked to leak DNS injector memory:

0000	0120	0001	0000	0000	0000	03	w	w	w	06
g	o	o	g	l	e	ff	c	o	m	00

But the first patch overlooked one of the exit conditions in the parsing loop. A query with QTYPE and QCLASS, and with a final label length prefix that exceeded the 127-byte threshold, still caused the parser to think the query was larger than it really was. A slightly modified probe format still worked to elicit the contents of memory:

0000	0120	0001	0000	0000	0000	03	w	w	w	06
g	o	o	g	l	e	03	c	o	m	ff	0001	0001

We named the pre-patch and post-patch vulnerabilities Wallbleed v1 and Wallbleed v2 respectively. We have used Wallbleed v1 probes in most of the experiments described in this paper. After the patch, we were able to resume experiments using modified probes, until Wallbleed v2 was finally patched in March 2024. With Wallbleed v2, only maximum-length overflows were possible: `ff` for a label length worked, but `20` did not. We found that the shortest domains, like `3.tt`, no longer worked as triggers, and therefore did later experiments with `te.rs`, the next shortest effective domain.

D. Other details of injection triggering

Here we comment on a few other details of the conditions to trigger injection. Note that there are other kinds of DNS injector in the GFW [8], [9], with its own blocklist and implementation quirks.

The injector defaults to type A responses. The DNS injectors respond only to queries whose QNAME matches a certain blocklist. The vulnerable injector injects type AAAA responses to type AAAA queries, and type A responses to queries of all other types.

The injector works on both IPv4 and IPv6. The UDP datagrams that carry DNS queries may be sent over IPv4 or

[2] We did this experiment on November 3, 2021. The name `3.tt` stopped triggering injection on August 7, 2023: see Section IV.

[3] There is a minor subtlety here. With names as short as these, it is technically possible to omit the final `00` byte, which otherwise is needed to terminate the flattened name string parsed from the query. The injector seems to zero the 18th byte of the destination buffer before copying the query into memory, so queries that are only 17 bytes long effectively have an implicit null terminator. As the DNS header takes up 12 bytes, this trick only works for QNAMEs as short as 5 bytes. But because the first leaked byte is a constant `00` in this case, shortening the QNAME from 6 to 5 bytes does not increase the number of informative bytes leaked. See Appendix B for an algorithmic description of this and other low-level details.

TABLE I: Experiment timeline and vantage points. In total, we used three VPSes in Tencent Cloud (TC, Beijing) (AS45090), one machine at the University Colorado Boulder (Scan, CO) (AS104) and one machine at the University of Massachusetts Amherst (Long, MA) (AS1249).

Experiments	Time Span		Duration	CN Hosts	US Hosts	Sections
Characterization	Oct. 2, 2021 – Feb. 10, 2022		4 months	1 (TC, Beijing)	1 (Long, MA)	§III-B
Re-characterization	May 9 – Sep. 10, 2023	& Feb, 2024	5 months	1 (TC, Beijing)	1 (Long, MA)	§III
Longitudinal	Nov. 21, 2021 – Apr. 16, 2024		2 years	3 (TC, Beijing)	1 (Long, MA)	§IV
Seeing Our Own	Aug. 12 – Sep. 8, 2023	& Mar. 13, 2024	4 weeks	1 (TC, Beijing)	1 (Scan, CO)	§V
Internet Scan	Jun. 25 & Aug. 23, 2023	& Mar. 6, 2024	3 days	–	1 (Scan, CO)	§VI
Patching Behavior	Sep. 6 – Nov. 7, 2023	& Mar. 6 – Apr. 16, 2024	3 months	2 (TC, Beijing)	2 (Scan & Long)	§VII §III-C

IPv6; the injector responds to either, forging an IPv4 or IPv6 response as appropriate for the query. (Here we are referring to the IP version over which the query is sent, not the QTYPE of the query. A query sent over IPv4 may request an IPv6 address and vice versa.) On May 9, 2023, we sent Wallbleed probes for the QNAME `ff|g|o|o|g|l|e|.|s|m|00` from a VPS in DigitalOcean (San Francisco, AS14061) to an IPv6 host in Alibaba Cloud (Beijing, AS37963) and to a non–DNS server 2400:dd01:103a:4041::101 in China. In both cases, we got an injected DNS response containing leaked memory. However, we could not trigger DNS injection in the other direction, sending queries from the VPS in China to the VPS or other IPv6 addresses in the US. This is likely because the injector was not deployed on the paths from our VPS in China to the foreign destinations we tested.

Only destination port 53 is looked at. On May 9, 2023, we sent queries for `google.sm` from the VPS in the US to our VPS in China, varying the UDP destination port over every value between 0 and 65535. Only queries sent to port 53 resulted in injections. This observation is consistent with prior findings of Lowe et al. in 2007 [21 §6.4] and Anonymous et al. in 2020 [8 §2.1].

IV. What information is leaked?

To better understand what information is leaked from the vulnerability, we conducted a longitudinal measurement and collected data for two years, from November 21, 2021 to November 29, 2023. Table I summarizes this experiment, as well as those of later sections.

Based on the observations in Section III-B, we designed the following *Wallbleed probe* to trigger the vulnerability:

`0000|0120|0001|0000|0000|0000|01|3|ff|t|t`

The probe is a query for `3.tt`, but truncated before the terminating empty label of QNAME (omitting the QCLASS and QTYPE fields), and with the `tt` label length prefix increased from `02` to `ff`. (Per footnote 3, no final `00` label is needed for a QNAME this short.) As explained in Section III-B, this probe causes a leak of 124 bytes of memory.

Experiment setup. We sent Wallbleed probes from a host in a US university to an IP address in China. The address in China was a VPS under our control in Tencent Cloud (AS45090). We varied the UDP source port of probes over a range of 1,000 port numbers (10001 to 11000), as prior work has suggested that the source port number may affect DNS injection [22]. We sent probes at a rate of 100 packets per second (pps) and collected 5.1 billion *Wallbleed responses* over two years.

Query names. The QNAME we started with, `3.tt`, was evidently removed from the injectors' blocklists and stopped eliciting injection responses on Monday, August 7, 2023 at 11:04:01 (China Standard Time, UTC+8). We changed the QNAME to `4.tt`, another short name from Section III-B.

A. Wallbleed leaks network traffic

Looking at samples of the 124-byte leaked fragments of memory, it is immediately clear that they include *snippets of network traffic*. These snippets originate, at least in part, in packets that pass by the injection device: in Section V we demonstrate recovery of packet payloads that we ourselves sent through the GFW. But the mix of protocols is different from what one would expect of a uniform sample of all traffic entering or exiting China.

After preliminary manual analysis of a sample of responses, we used regular expressions to search for common or sensitive strings. To reduce the risk of analyzing human-identifiable information, our program outputs only the number of matches. As shown in Table II, we find instances of UPnP, SSDP, HTTP, SMTP, SSH, and TLS, as well as potentially sensitive information such as HTTP cookies and passwords.

TABLE II: Counts of matches of regular expressions against 5.1 billion Wallbleed responses observed over two years.

Regular Expression	Note	Count	Rate
`ssdp:discover`	SSDP	184M	3.61%
`UPnP/IGD\xml`	UPnP	174M	3.41%
`(?s)[3-4]\xfftt.....-CONTROL`	(§IV-B)	121M	2.37%
`\x45\x00`	(§IV-A)	2.8M	0.05%
`uuid:WAN`	SSDP	34M	0.67%
`Host:␣`	HTTP	21M	0.41%
`(?i)Date:\s*...`	(§IV-C)	16M	0.31%
`\x7f\x00\x00`	(§IV-D)	2.8M	0.05%
`Cookie:␣`	HTTP	2.0M	0.04%
`RCPT␣TO`	SMTP	72.5k	0.0014%
`&key=`	URL	58.1k	0.0011%
`MAIL␣FROM`	SMTP	42.4k	0.0008%
`&password=`	URL	26.9k	0.0005%

It is remarkable that the memory contains application-layer protocols other than DNS. Since the injector responds to DNS queries only on UDP port 53 (Section III-D), we might expect

to see only DNS, or only UDP port 53 traffic; but in fact we see *a variety of protocols, including ones that typically run on different ports and transport protocols*. A noticeably large fraction consists of UPnP (Universal Plug and Play) and SSDP (Simple Service Discovery Protocol). UPnP uses HTTP—but there is more than an order of magnitude more UPnP than other forms of HTTP. The sample response in Section III is one such instance of UPnP. We extracted the HTTP Location header from 166 million UPnP snippets: in every case, the host part of the URL was a literal IP address in one of the private ranges of RFC 1918 [23]. Private addresses are consistent with UPnP and SSDP, which are normally used for service discovery in local networks. Nonetheless, it is difficult to explain why they appear at such a high rate in the memory of the vulnerable injector.

In addition to application-layer protocols, there are network-layer and transport-layer headers and packets. For instance, there are *IPv4 headers*. To find these, we first looked for the two-byte pattern `45 00` that commonly begins an IPv4 header, then (interpreting the bytes that follow as a header) filtered for a valid IP checksum. Table III shows the distribution of the `protocol` field in 181,834 IPv4 headers. TCP, UDP, and ICMP are the most common, with a long tail of 43 others.

TABLE III: The most common `protocol` fields in IPv4 headers. Only counts of more than 10 are shown.

Number	Protocol Name	Count
6	TCP	120,087
17	UDP	59,882
1	ICMP	1,735
50	ESP	38
0	IPv6 Hop-by-Hop Option	36
47	GRE	12

There were 7,743 cases where an IP header was followed by a TCP header and enough data to form a complete TCP segment, having consistent length fields and valid IP and TCP checksums. TCP headers have flags and port numbers, from which we may heuristically infer which of the two IP addresses in the IP header is the server, and which is the client. To avoid analyzing human-identifiable information, we anonymized IP addresses into two coarse categories: *private* (RFC 1918) and *public*. We then counted the proportions of client/server and private/public; the results are shown in Table IV.

TABLE IV: Client/server and private/public flows inferred from 7,743 complete TCP segments.

Client Address	Server Address	Count
Private	Private	384
Private	Public	6,276
Public	Private	193
Public	Public	890

Since the DNS injectors monitor public Internet traffic, we expected that TCP segments recovered from their memory would have mostly public IP addresses; however only 11% of TCP segments are public-to-public. Most of them actually involve a private client and a public server. Because private IP addresses are not globally routable, one might suspect that they represent part of the GFW's internal traffic (which would be compatible with the observations about UPnP above). However, the limited size of the memory leak means we can count only fairly short TCP segments (up to 125 bytes). It is also a possibility that the TCP segments we see were encapsulated in a higher-level protocol like GRE, not directly routed past the middlebox.

B. The four "digest" bytes

At the beginning of the longitudinal experiment, the first 4 bytes of leaked data in Wallbleed responses were different from the others. They were generally more random-looking, which was especially apparent when a leak otherwise consisted of readable ASCII. Dissimilar byte sequences might be attributed to partially overwritten memory, but this was different: it was consistently the first 4 bytes,[4] and they did not contain fragments of network protocols as the other bytes did. We took to calling these bytes "digest" bytes, on the supposition that they represented a hash of the query packet, possibly for load-balancing purposes. (This is only a guess—we tried, but did not find a hash algorithm that reproduced the digest bytes.) Digest bytes disappeared from Wallbleed responses in two stages across 2022 and 2023.

Digest bytes were not actually random, but were determined by the contents of the DNS query, including its UDP 4-tuple. On February 15, 2022 (at a time when all Wallbleed responses had digest bytes), we sent Wallbleed probes with identical payloads and source and destination IP addresses and ports. In all 114,717 resulting injections, the first 4 bytes were exactly `d8 fd d0 41`. Keeping the 4-tuple fixed and changing a bit in the payload, however, caused the digest bytes to change.[5] This may be compared to the injector process assignment of Section V-A, which depends on the 4-tuple but not the payload.

We measured the prevalence of digest bytes over time by looking for a particular string, `ACHE-CONTROL` (part of an HTTP Cache-Control header), that frequently appears at the beginning of Wallbleed responses (Table II). When digest bytes are present, the first 4 bytes of the string are overwritten. Figure 3 shows how digest bytes disappeared in two stages over nine months. When we began measurements, all responses had digest bytes. The first response that lacked digest bytes was on Saturday, September 3, 2022, at 01:31 (China Standard Time, UTC+8). After that, the presence or absence of digest bytes depended on the source port of the probe, with roughly half of ports eliciting digest bytes at a given point in time. The mapping of which ports caused digest bytes changed sporadically, but remained at a 50% fraction—we suspect this represents load balancing. After Thursday, June 8, 2023, at 15:33 (UTC+8), digest bytes almost completely disappeared.

[4]In the special case of a 17-byte query, including the probe used in this section, the digest bytes came after the initial `00` described in footnote 3.

[5]Changing a payload bit also changes the IP checksum, so digest bytes may have depended on IP and UDP headers only, or on headers and payload.

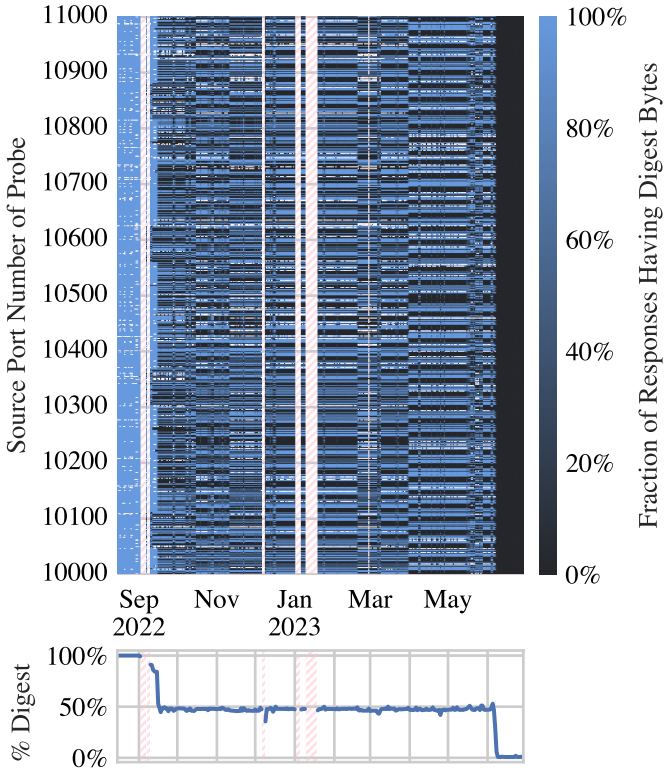

Fig. 3: The lower plot shows the rate of Wallbleed responses with digest bytes, averaged over all probe source ports in a day. Before September 3, 2022, all responses had digest bytes; after June 8, 2023, none did; and in between, half did and half did not. During the transition period, whether a given source port elicited digest bytes was consistent over short time spans. The upper plot shows the rate of digest responses by probe source port and day, which is always close to 0% or 100%.

C. How long bytes remain in memory

We estimated how long bytes tend to remain in memory by looking for naturally occurring timestamps, namely HTTP Date headers. These strings, in the format `Date: Wed, 21 Apr 2021 00:00:00 GMT`, indicate the time at which an HTTP response was generated. Figure 4 shows the distribution of the *age* of 16.3 million Wallbleed responses containing complete Date headers: the difference between when a response was received and the timestamp encoded in its Date header. Most Date headers are from the recent past: 75% are between 0 and 5 seconds old, and 7% are older. About 10% are nominally almost exactly 8 hours in the future relative to the time of capture, which is likely a result of servers wrongly reporting local time as UTC.

In Section V-A we conduct a similar memory-age experiment, using our own deliberately placed timestamps.

D. Inferring the GFW's internal architecture

Leaked memory occasionally contained what looked like x86_64 pointers. These are 64-bit values in little-endian byte order, whose most significant 16 bits are zero, and which

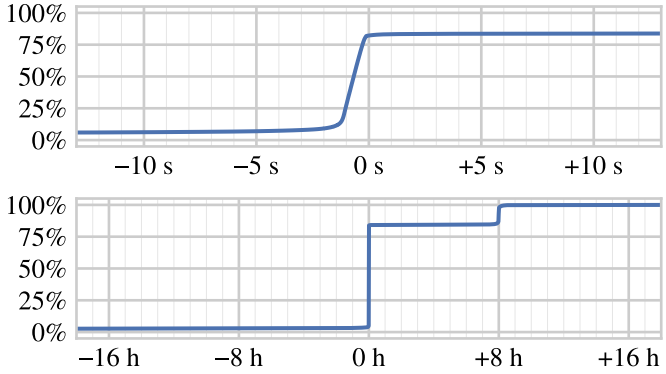

Fig. 4: Cumulative distribution of HTTP Date timestamps relative to the time of capture. The upper plot has a scale of seconds; the lower a scale of hours. Most timestamps are less than 5 seconds old. Time zone errors make some appear to be 8 hours in the future.

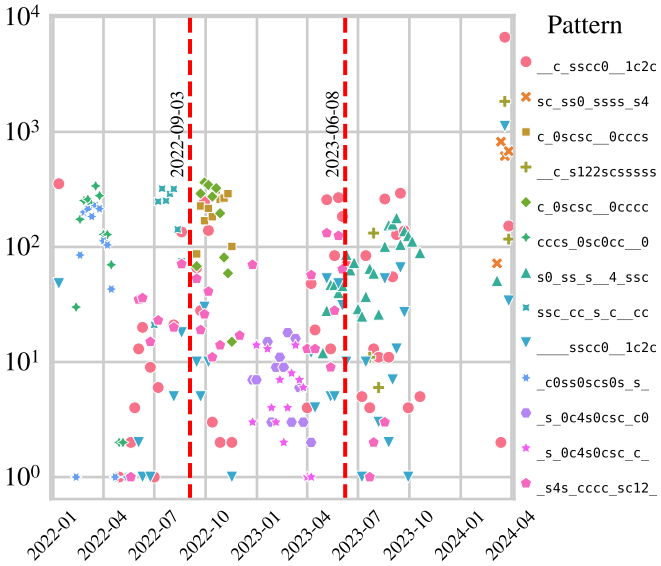

Fig. 5: The counts of common stack frame patterns in leaked memory seen weekly over time. '`s`' and '`c`' correspond to stack and code addresses respectively, and digits to specific common 64-bit values we observed. Red vertical lines indicate when we observed changes in the digest-byte pattern (Figure 3).

lie in conventional address ranges. On Linux, the typical address range of stack pointers is `0x00007f0000000000`–`0x00007fffffffffff`, and that of code and heap pointers is `0x0000550000000000`–`0x000056ffffffffff`.

A typical stack from on Linux contains a stack address followed by a code address (corresponding to a saved frame pointer and return address respectively). We looked for these patterns in our leaked payloads. We found 70,497 examples, and noticed several common patterns. We created pattern templates based off the 14 64-bit words present in each payload. For example, a stack address (a 64-bit value in the

typical Linux stack pointer range) is replaced with the single character ‘s’. In this way, code pointers (‘c’), and common numbers including zero (‘0’), −128 (‘1’), 22 (‘2’), and 4 (‘4’) are replaced, and remaining unlabeled words are converted to ‘_’. This yields 3,559 unique patterns, the most frequently occurring of which we plot in Figure 5.

The two red lines indicate the stages of digest byte transition from Figure 3. The first line, on September 3, 2023, coincides with a shift to the most seen stack frame patterns; the second, on June 8, 2023, does not show as clear a pattern change. We were not able to draw more concrete conclusions from these changing patterns though; they may be purely coincidental.

The stack frames we see are congruent with Linux stack frames with ASLR enabled, as indicated by a given pattern seeing randomization in a subset of bits: the least significant 12 bits are consistent across stack/code pointers, corresponding to the consistent offset in a 4 KB page. In some stack frames, we also observe what appear to be glibc stack canaries [24], indicated by a random value whose least significant 8 bits are set to 0 preceding a stack/code address pair.

We also observe *sequences of x86_64 instructions*, such as function prologues. We believe these to be code that the GFW sees on the network, not the code of the GFW itself, for two reasons. First, it is implausible that instructions would leak in a stack-based memory disclosure, as Linux clears pages before allocation and does not allow executable code in writable pages. Second, we also observe x86_64 code in traffic seen on a university network tap, which appears to be Microsoft code updates that send (signed) plaintext binaries [25].

V. Seeing our own traffic

In Section IV-A we saw evidence that Wallbleed leaked at least some network traffic, even non-DNS traffic, that passed through an injection device. Here we confirm that fact with a dedicated experiment. We sent our own tagged traffic across the border into China, and were later able to recover a fraction of it in Wallbleed responses.

Tagged traffic was recoverable only within a few seconds of its being sent. The recovery rate was low, and varied by time of day. Injection devices are internally divided into multiple independent processes, which we reveal using a previously undocumented side channel in the ordering of injected false IP addresses. Each process has its own memory: recovery of past traffic is possible only when a Wallbleed probe happens to be assigned to the same process. The assignment of packets to processes is deterministic, and depends on (at least) the source port of the probe. Probes sent over IPv6 may recover traffic originally sent over IPv4, and vice versa.

A. Timestamped magic sequence probes

We developed a new probe for this experiment. The *magic sequence probe* is a UDP packet, sent to port 53, whose 40-byte payload is two copies of the 20-byte sequence:

G	F	W	B	l	e	e	d	*exp*	*pkt*	*rep*
timestamp										

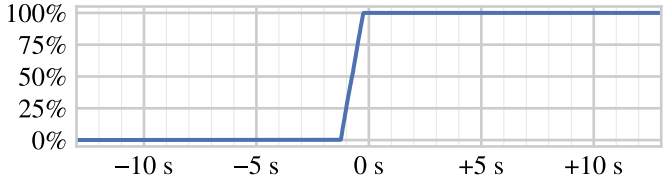

Fig. 6: Cumulative distribution of the difference between the timestamp stored in a magic sequence and when we recovered it in a Wallbleed response. The graph shows the distribution of 3,521 Wallbleed responses with magic sequences, collected between August 12 and September 8, 2023. The range of time differences is −10.19 s to −0.23 s.

where *exp* is an experiment ID, *pkt* is an incrementing packet ID, *rep* is 0 for the first copy of the sequence in a probe or 1 for the second, and *timestamp* is an epoch timestamp. The fixed string “GFWBleed” and unique IDs make it easy to identify magic sequences in Wallbleed responses. The timestamp lets us estimate how long a recovered magic sequence was kept in memory. While magic sequence probes use UDP and destination port 53, they are not DNS in structure.

At the same time as sending magic sequence probes, we sent Wallbleed probes, as in Section IV, to recover the sequences we were trying to place in memory. We sent the probes from a US university to a destination in China between August 12 and September 8, 2023 (Table I). The destination host was different from the one used in Section IV to avoid potential interference between the two experiments. We sent magic sequence probes at an average rate of 30 pps, from a single source port 10000. We sent Wallbleed probes at 100 pps from 199 source ports in the range 20001–20199. The choice to use a single source port for magic sequence probes would turn out to be significant, as it helped reveal the existence of discrete injector processes. We collected 3,521 Wallbleed responses containing magic sequences.

Recovered traffic is usually less than 1 second old. Figure 6 shows the difference between the timestamp encoded in a magic sequence probe and when it was recovered in a Wallbleed response. As with the HTTP Date timestamps of Section IV-C, traffic was short-lived in the injector’s memory: 99% of recovered magic sequences were less than 1.5 s old. The uniform slope between −1 s and 0 s is an artifact of the one-second granularity of *timestamp*. Unlike the HTTP Date experiment, here there is no possibility of time zone confusion.

The likelihood to recover traffic varies in a daily cycle. Figure 7 shows the number of Wallbleed responses containing magic sequences at each hour of the day, over 28 days. Though we sent Wallbleed probes and magic sequence probes at a constant rate, the number of probes recovered per hour varies in a 24-hour cycle, with a peak between 04:00 and 05:00 and a trough between 22:00 and 23:00 (China Standard Time, UTC+8). This is consistent with the *inverse* diurnal pattern of Internet traffic volume in China: the more traffic the injector handles, the less likely we are to observe our own packets.

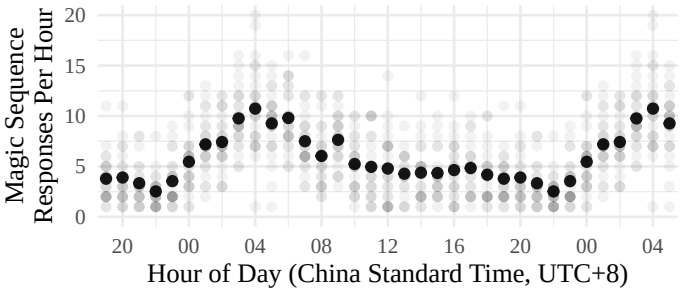

Fig. 7: The likelihood of observing a magic sequence depends on the time of day. The faint background points represent the number of Wallbleed responses containing a magic sequence received during every hour of the day, over four weeks starting on August 14, 2023. The dark foreground points are averages of corresponding hours across all 28 days.

Packets have consistent alignment in memory. When we recover a magic sequence, we do not get all 40 bytes in full. Almost always, the beginning is overwritten by the bytes of the Wallbleed probe that triggered the response. With the Wallbleed probe from Section IV, the first 18 bytes are overwritten and the last 22 bytes are intact. It is likely that the injection device aligns the first byte of packets at consistent locations in memory. Other observations support this hypothesis: in Section IV-B, we took advantage of the alignment of a common `ACHE-CONTROL` string to test for the presence of digest bytes.

Only a subset of source ports *ever* saw a magic sequence. We sent magic sequence probes from a single source port (10000). Though we sent Wallbleed probes from 199 different source ports (20001–20199), only 64 source ports ever recovered a magic sequence. (Those that did recovered 55 magic sequences on average.) Further investigation led us to believe that each DNS injection device consists of *multiple independent processes*, each with its own memory buffer, and that packets are deterministically assigned to a process according to features that include the source port. (But not the payload, because magic sequence probes had variable payloads. This contrasts with the digest bytes of Section IV-B, which did depend on the payload.) Only when a Wallbleed probe is assigned to the same process as the original magic sequence probe does it have a chance of recovering it. (This could explain the horizontal bands in Figure 3: for a time, half of processes used digest bytes and half did not.) In the next subsection, we show more evidence for the multiple-process hypothesis, in the form of a previously unknown side channel in the fake IP addresses of injected DNS responses.

B. The ordering of phony IP addresses

Previous research has shown that the GFW's DNS injection draws fake response IP addresses from a fixed pool—and that different subsets of the pool are used, depending on what name is queried [8 §3.2], [9 §5.3]. What has not been appreciated, before now, is that the pools are also *ordered* and *cyclic*. When probed at a high enough rate (around 100 queries per second or more—much greater than an injector's natural injection rate), using a consistent query name and source/destination IP address and port tuple, injected responses repeatedly cycle through IP addresses in the same order (with occasional gaps where the injector responded to queries from other users). By repeated probing, it is possible to get multiple copies of the sequence, reconcile the gaps, and recover the complete ordered list of false IP addresses for a given query name. A sample ordered list of 592 IP addresses for the query name `4.tt` appears in Appendix A.

Choosing any IP address to be "first" in the cycle, we may build a reverse mapping from an IP address to its *index*. Independent of the Wallbleed leak, every DNS response reveals the injector's internal index variable at the time of injection. Figure 8 shows the index of the IP address contained in Wallbleed responses over a 45-second interval, when probed at a high rate from 199 source ports. We see not one, but three roughly linear sequences. They are cyclic: when one reaches the top, it wraps around to the bottom. The same source port consistently maps to the same sequence. To us, it looks like hash-based load balancing over multiple processes within the injector device. The input to the load balancing assignment includes a packet's UDP 4-tuple, but not its data payload (because the magic sequence probes' payloads are variable). Keeping the rest of the 4-tuple fixed, source ports fall into a handful of equivalence classes according to which injector process they are assigned to. This explains why only 64 of 199 source ports recovered magic sequences: those are the ones that happened to be assigned to the same process as the magic sequence probes with source port 10000.

C. IPv4 and IPv6

Wallbleed provides a way to tell if IPv4 and IPv6 packets are processed on the same GFW nodes or different ones. If we send a unique payload over IPv4 past the GFW, and see parts of that payload in leaked memory from IPv6-based Wallbleed queries, then we know that there are nodes that process both IPv4 and IPv6 in the same memory.

We assembled a set of IPv6 prefixes that geolocate to China from MaxMind's GeoLite2 country code database [26] (downloaded March 12, 2024), excluding prefixes that were not routed, based on RouteViews BGP data (downloaded March 13, 2024). We sent a Wallbleed v2 probe to 8 random addresses in each IPv6 prefix. If at least 6 responded with a Wallbleed leak, we kept the prefix. We sampled these 610 IPv6 prefixes to obtain 133 k random IPv6 addresses that have a high likelihood of passing a GFW node. For IPv4 addresses, we randomly sampled 126 k IPv4 addresses that responded to an IPv4-wide ZMap scan, conducted March 6, 2024.

To each IPv4 and IPv6 address, we sent a *needle*: a UDP port 53 packet with a 900-byte payload consisting of a repeated sequence of an 8-byte string, 2-byte experiment ID, and 4-byte index that identified which IP address we sent the needle to. In parallel, we sent Wallbleed v2 probes to each address at a speed of 50 packets per second, and collected the responses

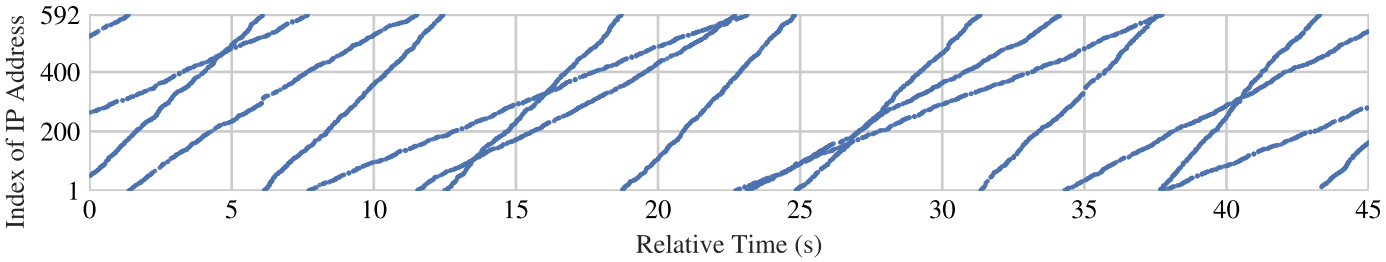

Fig. 8: A 45-second sample of Wallbleed responses received on August 23, 2023, the result of probing at a high rate from 199 different source ports. The IP address in each response has been reverse-mapped to its index (from 1 to 592) in the ordered list of Appendix A. The indices are not random, but form three distinct cyclic sequences—each source port consistently mapping to one of the three. Each sequence represents a process within the DNS injector, with its own address list iterator and memory allocation. Only 64 of 199 source ports mapped to the right process to recover the magic sequence probes of Section V-A.

to see if any contained previously sent needles. We repeated this process five times over 80 minutes.

There were 70 instances of one address receiving a Wallbleed-leaked payload containing a needle originally sent to a different address. Of these, 12 leaked from an IPv4 needle to an IPv4-probed address, 47 leaked IPv6-to-IPv6, 8 IPv4-to-IPv6, and 3 IPv6-to-IPv4. The presence of IPv4-to-IPv6 and IPv6-to-IPv4 leaks demonstrates that Wallbleed-vulnerable DNS injectors process both IPv4 and IPv6 traffic in the same memory space.

VI. IP Addresses Affected by Wallbleed

Wallbleed-prone DNS injectors formed part of the Great Firewall of China. Did these injectors affect every part of China, or anywhere outside China? How many IP addresses might have had their traffic pass through a vulnerable injector, and thus potentially be leaked? We did IPv4-wide scans from the outside of China to answer these questions. Both Wallbleed v1 and v2 affected IP addresses everywhere in China, consistent with the hypothesis of deployment of DNS injection at the network border. In many cases, even probes sent from the US to a place outside China got Wallbleed injections, because of network paths that transit the border.

A. IPv4-wide scan

We used ZMap [27] to scan the public IPv4 address space from a US university. To discover IP addresses affected by Wallbleed v1, we sent this payload to UDP port 53:

0000	0120	0001	0000	0000	0000	01	4	10	t	t

The payload is designed to elicit overflow from Wallbleed injectors, with only a small amount (14 bytes) of overflow to confirm the vulnerability. As is explained in footnote 3, this very short QNAME does not require a trailing `00` to be effective. We sent packets at a rate of 250 Mbps, and the scan took three hours.

We chose the name `4.tt` because it is unlikely to be on the DNS blocklists of countries other than China. As late as November 2020, `4.tt` was a Chinese-language gambling site.[6] (Gambling is one of the topics blocked by the Great Firewall [28 Art. 15], [9 §4.2].) The name no longer resolves to an IP address, and has not since at least July 2023. Using a China-focused and defunct name in our scans reduces the chance of triggering DNS injectors in other countries.

To discover IP addresses affected by Wallbleed v2, we sent the following payload to UDP port 53:

0000	0100	0001	0000	0000	0000	02	t	e	02	r
s	ff	0001	0001							

As introduced in Section III-C, `te.rs` was the shortest effective QNAME for Wallbleed v2, and the label length prefix had to extend past a constant threshold in the parser.[7]

Limitations. We ran the scans just three times: on June 25, 2023 and August 23, 2023 for Wallbleed v1, and on March 6, 2024 for Wallbleed v2. We scanned from one host in the US: other locations with different network paths to China might find different results. The results of this snapshot study reflect routing patterns at the time of the scan, and we cannot say how they may change over time. Similar injector middleboxes—with or without Wallbleed-like vulnerabilities—may exist in other countries, but our scans would not have found them, as we used a China-specific blocked domain.

B. Analysis of Wallbleed responses

Unless stated otherwise, the analysis in this section is based on the scan of August 23, 2023. The results of the June 25, 2023 scan for Wallbleed v1 and the March 6, 2024 scan for Wallbleed v2 were qualitatively similar.[8] The scan elicited 248.3 million responses from 245.4 million distinct IP addresses. 2.17 million IP addresses had more than one response, as many as 20,270 in one case, which may have been the result of routing loops [29], [30].

We used a two-step filter to separate Wallbleed injections from other responses. First, we filtered for responses whose answer section contained a false IP ad-

[6] https://web.archive.org/web/2020*/http://4.tt/

[7] We failed to limit the amount of overflow with Wallbleed v2 probes as we did with Wallbleed v1, which we might have done by adding a prefix to the QNAME to bring its length close to the injectors' maximum length threshold.

[8] Of the 1,157,694 /24 subnets with at least one responsive IP address in any scan, 10,181 (0.9%) responded to a v1 scan only, 27,351 (2.4%) responded to the v2 scan only, and 1,120,162 (96.8%) responded to both v1 and v2.

dress known to be used by Wallbleed injectors. To be precise, we kept responses that ended in a resource record of the form `c00c 0001 0001` *TTL* `0004` *a b c d* (type A), or `c00c 001c 0001` *TTL* `0010` *a b c d e f g h* (type AAAA), where *a.b.c.d* or *a*:*b*:*c*:*d*:*e*:*f*:*g*:*h* is one of the IP addresses in Appendix A. (Both type A and type AAAA responses are possible, though the probe did not specify a QTYPE.) Next, we filtered for responses beginning with the byte pattern `0000 8180 0001 0001 0000 0000 01 4 10 t t`; that is, a response for the QNAME and ID field of the probe, and flags equal to `8180`, as is characteristic of the affected injectors.

After filtering, there remained 244,911,941 responses (98.6% of all responses) from 242,442,549 distinct IP addresses that were definite Wallbleed injections. Table V shows the distribution of UDP payload lengths and DNS answer resource record types.

TABLE V: UDP length and DNS resource record type of Wallbleed responses in the Wallbleed v1 scan.

UDP Payload Length (Bytes)	# Responses	TYPE
52	244,881,083	A
64	30,837	AAAA
33	8	A
48	7	A
45, 46, 50, 51, 158	1	A
68	1	AAAA

In virtually all cases (99.99%), the response to our probe was a type A (IPv4) response of 52 bytes. 52 bytes is the expected length, given the label length prefix in the probe and the fixed size of the injector's answer section. In a small number of cases, the response was a type AAAA (IPv6) response of 64 bytes. There is an explanation for this effect: because our probe did not contain a QTYPE field, the injector took the QTYPE from bytes in memory located just after the probe. The injector defaults to type A responses, but in the special case that the bytes corresponding to QTYPE have the value `001c`, the injector crafts a type AAAA response instead.

C. Analysis of responding IP addresses

We used IP geolocation and IP-to-ASN mapping to find the location of IP addresses for which a Wallbleed response was received in the horizontal scans (after filtering out non-Wallbleed responses as described in the previous subsection). Unsurprisingly, almost all are reported to be in China, and they represent every geographic region of the country. A minority of responding IP addresses are reported to be outside China (after cross-checking against multiple databases to reduce the chance of geolocation errors).

We looked up every IP address affected by Wallbleed responses in the country-level IP2Location LITE DB5 database [31] (June 30, 2023) and the CAIDA ASN database [32] (July 18, 2023). The 242 million IP addresses for which a Wallbleed response was received map to 32 countries or regions, and belong to 381 ASes with 554 different ASNs.

TABLE VI: The ASes with the greatest number of Wallbleed-affected IP addresses. All are located in China, according to a geolocation database. When an AS has multiple ASNs, we show the one with the most affected IP addresses.

AS Name	ASNs	# IPs
China Telecom	4134, ...	104.2 M
China Unicom Backbone	4837, ...	54.9 M
China Mobile	9808, ...	23.9 M
China TieTong	9394, ...	12.8 M
China Unicom	4837, ...	12.7 M
Alibaba	37963, ...	7.3 M
Tencent	45090, ...	5.2 M
China Networks IX	4847	3.7 M
CERNET	4538	3.1 M
Oriental Cable Network	9812	1.7 M

Table VI shows the top ten ASes by number of responding IP addresses, all located in China.

For finer granularity, we sampled 10,000 IP addresses that country-level geolocation placed in China, and looked them up in the city- and province-level IP2Location LITE DB5 database [31] (August 24, 2023). The sampled IP addresses represented all 22 provinces, 5 autonomous regions, and 4 municipalities of China. We therefore surmise that Wallbleed-prone DNS injectors affected the entire country, not only certain regions.

Just 110,676 (0.05%) IP addresses mapped to a country other than China in country-level geolocation. It is not implausible that addresses outside China should be affected, as DNS injection is known to affect network paths that merely pass through China in transit [33]. But because geolocation databases can be inaccurate [34 §6.2], we applied additional filtering to eliminate addresses that are less certain to be outside of China:

1) We used three different databases: MaxMind GeoLite2 city [26] (September 1, 2023), IP2Location LITE DB5 [31] (August 24, 2023), and IPGeolocation.io [35] (October 2, 2023). If an IP address mapped to China in any database, we discounted its entire /24 network.
2) We looked up each IP address in the ASN databases of Team Cymru [36] (October 2, 2023) and CAIDA [32] (June 27, 2023). When an ASN's country of registration was China, we also discounted its entire /24 network.

The filter is designed to be conservative, in that it errs on the side of placing IP addresses in China. 6,822 IP addresses remained after filtering. Table VII summarizes them by AS, and Figure 9 shows their geolocation.

Though there likely remain a few incorrect geolocations, it is clear that some traffic outside of China may have been exposed to the privacy risk represented by Wallbleed. In 2010, Sparks et al. observed that 109 regions are DNS-polluted, primarily due to GFW DNS injections on the transit paths to TLD servers [33 §4.4]. In 2021, hosts in Mexico were not able to reach `whatsapp.net` as the GFW injected forged responses to queries to the root DNS servers in China [37], [38].

TABLE VII: Networks outside China for which Wallbleed responses were received in horizontal scans from the US. Two scans are represented, one on June 25, 2023 and one on August 23, 2023. The table shows the ten ASes with the greatest number of affected IP addresses. In total, there were 104 non-Chinese ASes in 37 countries in the June scan, and 99 ASes in 31 countries in the August scan.

AS Name	ASN	CC	# Unique IPs	
			Jun.	Aug.
Dreamline	9457	KR	1,534	1,086
MASTER-7-AS	26380	AU	315	489
Anpple Tech	133847	MY	243	257
Chinanet Backbone	4134	HK	235	248
AZT	53587	US	186	168
Network Joint	133762	HK	63	61
HK Broadband	9269	HK	50	85
STACKS-INC-01	398704	HK	31	78
Viettel Group	7552	VN	31	30
Aofei Data	135391	HK	29	28

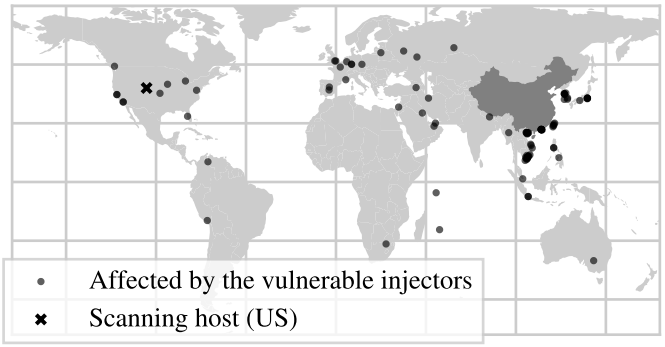

Fig. 9: City-level geolocation of IP addresses outside China for which a Wallbleed response was received, when scanning from our host in the US.

VII. Monitoring the Censor's Patching Behavior

We expected that the GFW would eventually patch the Wallbleed vulnerability. With a combination of continuous monitoring and China-wide scans, we captured the process of patching both Wallbleed v1 in September/October 2023, and Wallbleed v2 in March 2024.

Experiment setup. For continuous monitoring, we sent Wallbleed probes and ordinary DNS queries for the same QNAME, at 100 pps, from the US to an IP address under our control in China. We used `4.tt` for v1 probes and and `te.rs` for v2 probes. The ordinary DNS queries acted as controls to distinguish patching of the vulnerability from the injector being offline or the QNAME being removed from the blocklist. If the injector stops responding to Wallbleed probes, but continues to respond to the normal probes without interruption, this is evidence that the censor can *hot-patch* the GFW with minimal downtime. On the other hand, if the injector stops responding to both for some time, and later resumes responding to normal probes only, then we can measure the downtime related to patching. Using a machine

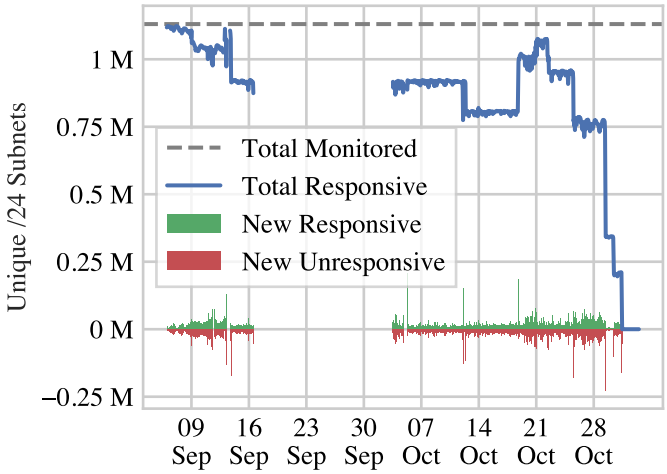

Fig. 10: We tracked the number of IPv4 /24 subnets that responded to Wallbleed v1 probes over time. We scanned 1,130,343 IP addresses (one per subnet) every 15 minutes between September 6 and November 7, 2023. We failed to collect data between September 17 and October 4, 2023. Wallbleed v1 was patched in two major stages: between September 6 and 14, 2023; and between October 22 and November 1, 2023.

in UMass Amherst, we did continuous monitoring of Wallbleed v1 between September 6 and November 7, 2023, and of Wallbleed v2 between March 6 and April 16, 2024.

We also did scans of a sample of about one million addresses in China. These were designed to test whether patching would happen at different times in different regions, or simultaneously across the country. We selected one representative per /24 subnet from the 215 million responsive IPv4 addresses discovered in the IPv4 scan of Section VI, yielding 1,130,343 IP addresses. We used ZMap [27] to send a Wallbleed probe to each of these IP addresses every 15 minutes. We conducted these scans from CU Boulder, between September 6 and November 7, 2023 for Wallbleed v1, and between March 28 and April 16, 2024 for Wallbleed v2.

Experiment results. Figure 10 shows the number of /24 subnets that responded to Wallbleed v1 probes in ZMap scans, as well as the hourly churn rate: the number of IP addresses that were responsive in one hour but not the next, or vice versa. We aggregated responding IP addresses by hour to reduce false negatives caused by packet loss.

There are some variations in the response rate leading up to October 23, when the Wallbleed vulnerability was patched over about a week. Starting on October 23, we observed discrete steps down in response rate as the vulnerability was progressively patched. The last three steps occurred on October 30 (Monday), October 31 (Tuesday), and November 1 (Wednesday) at the same time each day: between 10:00 and 12:00 (China Standard Time, UTC+8). After 12:00 on November 1, we no longer saw Wallbleed v1 responses for any IP addresses we scan. We examined the IP addresses that

transitioned to unresponsive in the step on October 30. 86% of the 39 k addresses were part of a /20 subnet that no longer responded, indicating that the discrete steps corresponded to large blocks of IP addresses changing in tandem, rather than a more randomized, load-balancing style of update.

Wallbleed v2 was completely patched by March 28, 2024. Unfortunately, we only captured the last 60 minutes of the patching process, in four horizontal scans. Like Wallbleed v1, Wallbleed v2 was patched in discrete steps. We isolated the time of final patching of Wallbleed v2 to between 16:01:30 and 16:16:30 (China Standard Time, UTC+8) on March 28, 2024 (a different time of day than v1).

In the last one-hour capture, 42,084 IP addresses elicited Wallbleed v2 responses. Interestingly, 33,779 (80.3%) of these addresses belong to AS4538 (CERNET, the China Education and Research Network Center), along with a long tail of 49 ASes that belong to China Mobile and various universities in China. This observation supports the hypothesis that CERNET maintains a subset of the national GFW infrastructure. The DNS injector in CERNET had the Wallbleed v2 vulnerability in common with the rest of the GFW, suggesting unified management and coordinated patching. Meanwhile, its distinct patching schedule demonstrates a degree of independence in its operation and maintenance.

VIII. Related work

DNS injection by the Great Firewall is one of the oldest and most-studied forms of Internet censorship. The earliest documentation we know of is in two independent studies from 2002, one by Dong [39] and one by Zittrain and Edelman [40], both of which found that a single bogus IP address was used in all injected responses. In 2009, gfwrev discovered two types of DNS injector in China with distinct fingerprints, and documented another seven response IP addresses in addition to the one that had been used in 2002 [13]. In 2014, Anonymous et al. analyzed IP ID and TTL patterns in injected responses to infer the existence of 367 separate injection processes, each injecting at a rate of between 0 and 60 fake DNS responses per second [7 §7]. The number of bogus IP addresses in use had grown to at least 174 by 2016 [41], [42]. Anonymous et al. distinguished the fingerprints of at least three DNS injectors in 2020 [8]. Large-scale measurements by Hoang et al. in 2021 showed that tracking changes in the GFW's DNS domain blocklists can help in understanding censorship trends in China [9].

The past work that most resembles, and indeed inspires, our own is gfw-looking-glass.sh, a one-line shell script posted by klzgrad from gfwrev in 2010 [5], [6]. To the best of our knowledge, it was the first memory-dumping vulnerability in the GFW. DNS queries with a name truncated after the first byte of a 2-byte compression pointer caused the GFW's DNS parser to treat nearby memory as part of the name, and leak it back in the injected response. This vulnerability was fixed prior to our discovery of Wallbleed. The script incidentally demonstrated that a query name containing an embedded dot character, `06 w u x . r u`, was treated the same as one correctly split into separate labels, `03 w u x 02 r u`, indicating that the GFW, at that time too, serialized the name to a dotted string before matching it against a blocklist, rather than matching on structured labels. In 2014, klzgrad found that the GFW's DNS injector had ceased to interpret compression pointers, opening opportunities to evade DNS injection with queries that used pointers in unusual ways [43].

Wallbleed was independently discovered by Sakamoto and Wedwards in 2023 [44]. They analyzed the leaked data, inferred the characteristics of the GFW's processes, and proposed several attacks leveraging this vulnerability. Apart from confirming their observations, we developed the study of Wallbleed further with longitudinal and Internet-wide measurements of more than two years since October 2021. We uncovered the root cause of Wallbleed, reconstructed the parsing logic in C code, used a novel side channel to identify individual processes in the vulnerable injector, examined affected IP addresses, and, after the first incomplete patch of November 2023, found the Wallbleed v2 vulnerability.

Wallbleed is named like other similar memory disclosure vulnerabilities. Heartbleed, a vulnerability in OpenSSL, allowed clients to leak up to 64 KB of a TLS server's memory at a time [1]. Cloudbleed was a vulnerability in an HTML parser used on edge servers of the Cloudflare content delivery network in 2017 [3], [4]. Similarly, Ticketbleed documented a vulnerability in F5 middleboxes [2].

IX. Ethics

Three main ethical considerations arise in this research. The first is the handling of experimental data, such as what we collected over two years in our longitudinal experiment. If, as we contend, Wallbleed represented a privacy risk to the users whose traffic passed through vulnerable injectors, then the storage and analysis of leaked data require sensitivity and care. The second is whether, or under what circumstances, it is okay to exploit a security vulnerability in a system that may itself be regarded as a hostile network attacker [10], [11]—in this case the GFW. The third is how to approach disclosure.

A. Data handling

The experiment in Section V demonstrates that at least *some* of the data exposed by Wallbleed to third parties originated in traffic transiting the firewall. This presents a privacy concern: network traffic may contain sensitive information such as usernames, passwords, or web requests. We submitted our research plan to our institutional review board (IRB), which exempted the research as not involving human subjects. Below we detail our considerations and safeguards for this data.

Data collection. There is an unavoidable trade-off between reducing data collection and being able to do meaningful analysis. Once the Wallbleed vulnerability was understood, leaking a single byte would have been sufficient to confirm its presence, but such limited measurement would not have allowed us to study the firewall's architecture or how Internet users were impacted. On-the-fly analysis of in-memory (rather

than stored) data would have allowed us to report some results, but we would not have noticed and would not have been able to analyze unanticipated changes, such as the gradual disappearance of "digest" bytes in Section IV-B. We therefore focused on a strategy of protecting collected data, rather than artificially limiting what was collected. Ultimately, after discussion within our team and with reviewers, we decided to delete the collected data upon publication of this work.

B. Ethics of exploitation

Exploiting a bug of this nature is ethically complicated. From a deontological perspective [45 §4.1], security researchers might decide to avoid exploiting vulnerabilities in systems they do not control under any circumstances, as doing so may have unintended and negative impacts that are difficult to predict. Alternatively, from a consequentialist perspective [45 §4.1], one must weigh the benefits of research against its risk of harms.

We identify two high-level sources of potential harm and negative effects in our research: (1) the data we collected, which may contain sensitive information, could leak; and (2) the probes we sent may cause the GFW, or other middleboxes or end hosts, to crash or malfunction. We have discussed the first source of risks in Section IX-A. Below we discuss how we manage the second source of risks.

Given that the system we exploit is itself considered by many to be a source of harm [10], [11], even if our experiments result in damage to the GFW, it will essentially reduce harm to more than a billion people by hampering censorship. In particular, any crash of the GFW is unlikely to impede network traffic. Past research has shown that GFW DNS injectors are on-path devices [7]–[9], [12], [21], [39], [46]–[48]; that is, they work by getting a mirrored copy of traffic, and are not themselves a link in the transmission chain. Finally, prior work has exploited vulnerabilities in other harmful systems like botnets and middleboxes in order to study those problematic systems [29], [49]–[52].

To minimize the risk of crashing other middleboxes and end hosts, we cautiously only sent traffic to hosts under our control during the first 18 months of experiments. Only after observing a lack of adverse effects did we start Internet-wide scans. Following best practices for Internet scanning [27], we limited the traffic volume to each host not under our control to only one UDP packet per 15 minutes. We hosted a web page at the source IP address of our scans, displaying a project description and explaining how to opt out of scanning. We received and honored one opt-out request in the course of the study.

C. Whether to disclose, and how

Disclosing a bug of this nature is also complicated. By reporting the vulnerability, are we ultimately "helping" the Great Firewall? There is also a trade-off to consider between immediate and delayed disclosure: remove the privacy risk to users now, or take time to gain a greater understanding of the censorship system, in order to perhaps avoid even greater risks and harm in the future?

We decided on a strategy of coordinated disclosure, but only after taking advantage of the opportunity occasioned by the vulnerability to learn as much about the DNS injection subsystem as possible. Two factors led us to the decision to eventually disclose. The first is the risk to the privacy of users. Once the unpatched bug was made public, it could be used by others who do not have regard for users' safety. The second is that the Wallbleed vulnerability does not reduce the effectiveness of the DNS censorship system. With Wallbleed fixed, the injectors carry on interfering with connections as before, but they do not do *more* of it.

This ethical calculus is specific to this situation. Under other circumstances we might come to a different decision. If there were an implementation error in the Great Firewall that caused it to fail to censor some fraction of connections, and otherwise did not increase risk to users, we would not be obligated to report it. Our allegiance is not to bug-fixing in the abstract, but to the security of users. We maintain that the only correct fix for a bug like Wallbleed is the removal of affected devices (i.e., the GFW injectors) from the network: the real "bug" is their very presence, not in the specific implementation errors they undoubtedly have. The incomplete patch in November 2023 that led to the Wallbleed v2 variant reinforces the point: as long as the injectors exist, they will pose a risk to users.

In the end, our decision to disclose was made moot by the patching of the vulnerability, before we were able to report the issue to CNCERT. This paper, too, forms part of our disclosure strategy: documenting and publicizing this vulnerability will draw more attention to the many dangers of censorship.

X. Lesson learned for future work

Our study provides a unique case study of the balance between protecting user data and utility of research data in understanding a system. While in hindsight it is possible to see areas where we could have chosen to collect less data (and thus reduced the risk of collecting personal information), we note that it is difficult to know the optimal boundaries of unstructured data ahead of time. For instance, we learned of the 4-byte "digest bytes" feature by studying a large number of full payloads. In hindsight, we might still have discovered this feature of the GFW by leaking only 4 bytes, so it may appear unnecessary to collect more than that. But without knowledge of the nature of this feature ahead of time, it would be difficult to know that 4 bytes would be sufficient. Likewise, when choosing how many bytes to leak, we are faced with a difficult trade-off: leak more bytes at the risk of collecting personal data (but potentially learn more about a yet-unknown feature of the GFW), or leak fewer bytes at the risk of learning less (but limit the potential collection of sensitive data). This type of trade-off should be considered carefully for all work, and we hope that by documenting our thought process, we inspire further discussion and debate among the research community.

IRB decisions. We were asked to push back on our Institutional Review Board's decision to mark our work as exempt, as reviewers felt that there were additional ethical considerations

of our work not captured by that decision. We agree that our work has complex ethical considerations, but disagree that these considerations fall explicitly within the bounds of an IRB, or that authors should be required to push back on IRB decisions they disagree with.

We were transparent in the protocol we submitted to the IRB, including that the collected data may contain third-party network traffic. This is an excerpt from our submitted protocol:

> "We have discovered that, in processing certain malformed DNS queries, the subsystem may include fragments of unrelated system memory in its injected response. In short, the firewall 'leaks' small fragments of memory, which may incidentally include other people's network traffic that passes by the GFW. The discovery is significant both for network security and for understanding the Great Firewall. While the contents of the leaked memory are unpredictable, it is possible that they may contain personally identifiable information, such as IP addresses. Therefore, we are seeking the IRB's guidance on how to proceed, and in particular, on whether this research requires full IRB review."

We recognize that IRB exemption is not the same thing as the IRB making an ethical determination about the work, nor does it necessarily mean the IRB feels there is no potential harm or ethical considerations needed. Rather, an exemption means that the IRB has determined it does not fall within a narrow definition of "human subjects research" [53 § 219.102]. To this end, we treated the data we collected as sensitive, and have deleted it prior to publication as a precaution against potential abuse. Nonetheless, we feel it is important for our community to understand the limitations of IRBs, and to avoid using them as a stand-in for ethical decisions.

XI. Conclusion

In this work, we present and study Wallbleed, a buffer over-read vulnerability in the DNS injection subsystem of the Great Firewall of China (GFW). We conducted longitudinal and Internet-wide measurements to understand the cause and implications of Wallbleed. We also revealed details of the GFW's internal architecture and operations that would not be possible to learn about without Wallbleed. Wallbleed exemplifies that the harm censorship middleboxes impose on Internet users goes even beyond the direct (and designed) harm of censorship: it can severely violate users' privacy and confidentiality.

Availability

To encourage future research and promote transparency and reproducibility, we have made the code, anonymized data, and additional context about our work and publishing process publicly available. For improved accessibility, we offer both English and Chinese HTML versions of the paper. The project homepage is at: https://gfw.report/publications/ndss25/en.

Acknowledgments

We are deeply grateful to several colleagues who wish to remain unnamed for their valuable contributions and guidance throughout the entire project. We are also thankful to klzgrad from gfwrev for their inspiring pioneering work in 2010, and for providing rounds of thoughtful comments in this study. In addition, we thank Alberto Dainotti, Ali Zohaib, Cecylia Bocovich, Diogo Barradas, J. Alex Halderman, Jakub Dalek, Jeffrey Knockel, Michael Carl Tschantz, Nadia Heninger, Philipp Winter, ppmaootc, Prateek Mittal, Xiao Qiang, and Zakir Durumeric. We also thank the anonymous reviewers for their helpful comments and guidance.

References

[1] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The matter of Heartbleed," in *Internet Measurement Conference*. ACM, 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2663716.2663755

[2] F. Valsorda. (2016) Ticketbleed (CVE-2016-9244). [Online]. Available: https://filippo.io/Ticketbleed/

[3] J. Graham-Cumming. (2017, Feb.) Incident report on memory leak caused by Cloudflare parser bug. [Online]. Available: https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/

[4] M. Prince. (2017, Mar.) Quantifying the impact of "Cloudbleed". [Online]. Available: https://blog.cloudflare.com/quantifying-the-impact-of-cloudbleed/

[5] gfwrev. (2010, Sep.) "gfw-looking-glass.sh: while true; do printf "\0\0\1\0\0\1\0\0\0\0\0\0\6wux.ru\300" | nc -uq1 $SOME_IP 53 | hd -s20; done". [Online]. Available: https://twitter.com/gfwrev/status/25220534979

[6] Anonymous. (2020, Mar.) GFW archaeology: gfw-looking-glass.sh. [Online]. Available: https://github.com/net4people/bbs/issues/25

[7] ——, "Towards a comprehensive picture of the Great Firewall's DNS censorship," in *Free and Open Communications on the Internet*. USENIX, 2014. [Online]. Available: https://www.usenix.org/system/files/conference/foci14/foci14-anonymous.pdf

[8] Anonymous, A. A. Niaki, N. P. Hoang, P. Gill, and A. Houmansadr, "Triplet censors: Demystifying Great Firewall's DNS censorship behavior," in *Free and Open Communications on the Internet*. USENIX, 2020. [Online]. Available: https://www.usenix.org/system/files/foci20-paper-anonymous_0.pdf

[9] N. P. Hoang, A. A. Niaki, J. Dalek, J. Knockel, P. Lin, B. Marczak, M. Crete-Nishihata, P. Gill, and M. Polychronakis, "How great is the Great Firewall? Measuring China's DNS censorship," in *USENIX Security Symposium*. USENIX, 2021. [Online]. Available: https://www.usenix.org/system/files/sec21-hoang.pdf

[10] Internet Society. (2023, Dec.) When is the Internet not the Internet? [Online]. Available: https://www.internetsociety.org/resources/internet-fragmentation/the-chinese-firewall/

[11] D. Anderson, "Splinternet behind the Great Firewall of China: Once China opened its door to the world, it could not close it again," *Queue*, vol. 10, no. 11, pp. 40–49, Nov. 2012. [Online]. Available: https://queue.acm.org/detail.cfm?id=2405036&doi=10.1145%2F2390756.2405036

[12] H. Duan, N. Weaver, Z. Zhao, M. Hu, J. Liang, J. Jiang, K. Li, and V. Paxson, "Hold-On: Protecting against on-path DNS poisoning," in *Securing and Trusting Internet Names*. National Physical Laboratory, 2012. [Online]. Available: https://www.icir.org/vern/papers/hold-on.satin12.pdf

[13] gfwrev. (2009, Nov.) 深入理解GFW：DNS污染. [Online]. Available: https://gfwrev.blogspot.com/2009/11/gfwdns.html

[14] Z. Chai, A. Ghafari, and A. Houmansadr, "On the importance of encrypted-SNI (ESNI) to censorship circumvention," in *Free and Open Communications on the Internet*. USENIX, 2019. [Online]. Available: https://www.usenix.org/system/files/foci19-paper_chai_update.pdf

[15] K. Bock, G. Naval, K. Reese, and D. Levin, "Even censors have a backup: Examining China's double HTTPS censorship middleboxes," in *Free and Open Communications on the Internet*. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3473604.3474559

[16] N. P. Hoang, J. Dalek, M. Crete-Nishihata, N. Christin, V. Yegneswaran, M. Polychronakis, and N. Feamster, "GFWeb: Measuring the Great Firewall's Web censorship at scale," in *USENIX Security Symposium*. USENIX, 2024. [Online]. Available: https://www.usenix.org/system/files/sec24fall-prepub-310-hoang.pdf

[17] K. Bock, iyouport, Anonymous, L.-H. Merino, D. Fifield, A. Houmansadr, and D. Levin. (2020, Aug.) Exposing and circumventing China's censorship of ESNI. [Online]. Available: https://github.com/net4people/bbs/issues/43

[18] A. Master and C. Garman, "A worldwide view of nation-state Internet censorship," in *Free and Open Communications on the Internet*, 2023. [Online]. Available: https://www.petsymposium.org/foci/2023/foci-2023-0008.pdf

[19] S. Nourin, V. Tran, X. Jiang, K. Bock, N. Feamster, N. P. Hoang, and D. Levin, "Measuring and evading Turkmenistan's internet censorship," in *The International World Wide Web Conference*. ACM, 2023. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3543507.3583189

[20] P. Mockapetris, "Domain names - implementation and specification," RFC 1035, Nov. 1987. [Online]. Available: https://www.rfc-editor.org/info/rfc1035

[21] G. Lowe, P. Winters, and M. L. Marcus, "The great DNS wall of China," New York University, Tech. Rep., 2007. [Online]. Available: https://censorbib.nymity.ch/pdf/Lowe2007a.pdf

[22] A. Bhaskar and P. Pearce, "Many roads lead to Rome: How packet headers influence DNS censorship measurement," in *USENIX Security Symposium*. USENIX, 2022. [Online]. Available: https://www.usenix.org/system/files/sec22-bhaskar.pdf

[23] R. Moskowitz, D. Karrenberg, Y. Rekhter, E. Lear, and G. J. de Groot, "Address allocation for private Internets," RFC 1918, Feb. 1996. [Online]. Available: https://www.rfc-editor.org/info/rfc1918

[24] hugsy, "Playing with canaries," Jan. 2017. [Online]. Available: https://www.elttam.com/blog/playing-with-canaries/#glibc-analysis

[25] M. Phaedrus, "Some technical details behind the mundane Windows update," 2022. [Online]. Available: https://great-computing.quora.com/Some-technical-details-behind-the-mundane-Windows-Update-https-www-quora-com-Does-the-Windows-update-use-HTTP-answer

[26] "MaxMind GeoLite2 geolocation database." [Online]. Available: https://dev.maxmind.com/geoip/geolite2-free-geolocation-data

[27] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *USENIX Security Symposium*. USENIX, Aug. 2013. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric

[28] State Council of the People's Republic of China, "互联网信息服务管理办法 (*Measures for the Administration of Internet Information Services*)," Sep. 2000. [Online]. Available: https://www.gov.cn/gongbao/content/2000/content_60531.htm

[29] K. Bock, A. Alaraj, Y. Fax, K. Hurley, E. Wustrow, and D. Levin, "Weaponizing middleboxes for TCP reflected amplification," in *USENIX Security Symposium*. USENIX, 2021. [Online]. Available: https://www.usenix.org/system/files/sec21-bock.pdf

[30] A. Alaraj, K. Bock, D. Levin, and E. Wustrow, "A global measurement of routing loops on the Internet," in *Passive and Active Measurement*. Springer Nature Switzerland, 2023, pp. 373–399. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-28486-1_16

[31] "IP2Location LITE IP address geolocation database." [Online]. Available: https://www.ip2location.com/database/ip2location

[32] CAIDA, "CAIDA AS to organization mapping dataset." [Online]. Available: https://www.caida.org/catalog/datasets/request_user_info_forms/as_organizations/

[33] Sparks, Neo, Tank, Smith, and Dozer, "The collateral damage of Internet censorship by DNS injection," *SIGCOMM Computer Communication Review*, vol. 42, no. 3, pp. 21–27, 2012. [Online]. Available: https://conferences.sigcomm.org/sigcomm/2012/paper/ccr-paper266.pdf

[34] Z. Weinberg, S. Cho, N. Christin, V. Sekar, and P. Gill, "How to catch when proxies lie: Verifying the physical locations of network proxies with active geolocation," in *Internet Measurement Conference*. ACM, 2018. [Online]. Available: https://www.contrib.andrew.cmu.edu/~nicolasc/publications/Weinberg-IMC18.pdf

[35] "IPGeolocation.io IP geolocation API." [Online]. Available: https://ipgeolocation.io/documentation/ip-geolocation-api.html

[36] Team Cymru, "Team Cymru IP to ASN lookup v1.0." [Online]. Available: https://asn.cymru.com/

[37] Q. Lone. (2022, Apr.) Detecting DNS root manipulation. [Online]. Available: https://labs.ripe.net/author/qasim-lone/detecting-dns-root-manipulation/

[38] Y. Nosyk, Q. Lone, Y. Zhauniarovich, C. H. Gañán, E. Aben, G. C. M. Moura, S. Tajalizadehkhoob, A. Duda, and M. Korczyński, "Intercept and inject: DNS response manipulation in the wild," in *Passive and Active Measurement*. Springer Nature Switzerland, 2023. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-28486-1_19

[39] B. Dong, "A report about national DNS spoofing in China on Sept. 28th," Dynamic Internet Technology, Inc., Tech. Rep., Oct. 2002. [Online]. Available: https://web.archive.org/web/20021015121616/http://www.dit-inc.us/hj-09-02.html

[40] J. Zittrain and B. G. Edelman, "Internet filtering in China," *IEEE Internet Computing*, vol. 7, no. 2, pp. 70–77, Mar. 2003. [Online]. Available: https://nrs.harvard.edu/urn-3:HUL.InstRepos:9696319

[41] O. Farnan, A. Darer, and J. Wright, "Poisoning the well – exploring the Great Firewall's poisoned DNS responses," in *Workshop on Privacy in the Electronic Society*. ACM, 2016. [Online]. Available: https://dl.acm.org/authorize?N25517

[42] P. Pearce, B. Jones, F. Li, R. Ensafi, N. Feamster, N. Weaver, and V. Paxson, "Global measurement of DNS manipulation," in *USENIX Security Symposium*. USENIX, 2017. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-pearce.pdf

[43] klzgrad. (2014, Nov.) DNS compression pointer mutation. [Online]. Available: https://gist.github.com/klzgrad/f124065c0616022b65e5

[44] Sakamoto and E. Wedwards, "Bleeding wall: A hematologic examination on the Great Firewall," in *Free and Open Communications on the Internet*, 2024. [Online]. Available: https://www.petsymposium.org/foci/2024/foci-2024-0002.pdf

[45] T. Kohno, Y. Acar, and W. Loh, "Ethical frameworks and computer security trolley problems: Foundations for conversations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5145–5162. [Online]. Available: https://securityethics.cs.washington.edu/

[46] M. C. Tschantz, S. Afroz, Anonymous, and V. Paxson, "SoK: Towards grounding censorship circumvention in empiricism," in *Symposium on Security & Privacy*. IEEE, 2016. [Online]. Available: https://www.eecs.berkeley.edu/~sa499/papers/oakland2016.pdf

[47] X. Xu, Z. M. Mao, and J. A. Halderman, "Internet censorship in China: Where does the filtering occur?" in *Passive and Active Measurement Conference*. Springer, 2011, pp. 133–142. [Online]. Available: https://web.eecs.umich.edu/~zmao/Papers/china-censorship-pam11.pdf

[48] Z. Wang, Y. Cao, Z. Qian, C. Song, and S. V. Krishnamurthy, "Your state is not mine: A closer look at evading stateful Internet censorship," in *Internet Measurement Conference*. ACM, 2017. [Online]. Available: https://www.cs.ucr.edu/~krish/imc17.pdf

[49] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: analysis of a botnet takeover," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 635–647. [Online]. Available: https://sites.cs.ucsb.edu/~chris/research/doc/ccs09_botnet.pdf

[50] A. Mirian, A. Ukani, I. Foster, G. Akiwate, T. Halicioglu, C. T. Moore, A. C. Snoeren, G. M. Voelker, and S. Savage, "In the line of fire: Risks of DPI-triggered data collection," in *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, 2023, pp. 57–63. [Online]. Available: https://arianamirian.com/docs/cset2023_fireye.pdf

[51] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamalytics: An empirical analysis of spam marketing conversion," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 3–14. [Online]. Available: https://www.icir.org/christian/publications/2008-ccs-spamalytics.pdf

[52] K. Bock, P. Bharadwaj, J. Singh, and D. Levin, "Your censor is my censor: Weaponizing censorship infrastructure for availability attacks," in *Workshop on Offensive Technologies*. IEEE, 2021. [Online]. Available: https://www.cs.umd.edu/~dml/papers/weaponizing_woot21.pdf

[53] U.S. Government, "Title 32 of the Code of Federal Regulations § 219.102: Definitions," 2024. [Online]. Available: https://www.ecfr.gov/on/2024-11-27/title-32/subtitle-A/chapter-I/subchapter-M/part-219/section-219.102

APPENDIX A
AN EXAMPLE ORDERED POOL OF FAKE IP ADDRESSES

Below are the ordered lists of 592 IPv4 and 30 IPv6 addresses used by the Wallbleed-affected DNS injectors when forging responses to A and AAAA queries, respectively, for the DNS name `4.tt`. The pools for other injectors and other query names may differ [8 § 3.2] [9 § 5.2]. When an injector process injects a DNS response, it takes the next IP address from its ordered list, cycling back to the beginning after reaching the end. This fact becomes evident when collecting injected responses at a sufficiently high sample rate (around 100 packets per second or more). The selection of a "first" address in each cycle is arbitrary.

We put the ordered lists to two uses: in Section VI-C, to filter Wallbleed-related DNS responses from other responses; and in Section V-B, to isolate the multiple independent processes within each injector device. Machine-readable versions are included with the data published alongside this paper.

1) 208.77.47.172
2) 173.252.88.133
3) 173.252.88.67
4) 108.160.170.45
5) 157.240.10.32
6) 174.36.196.242
7) 174.36.228.136
8) 174.37.154.236
9) 174.37.175.229
10) 174.37.54.20
11) 185.60.216.11
12) 185.60.216.50
13) 199.16.158.12
14) 199.16.158.182
15) 199.16.158.8
16) 199.16.158.9
17) 199.59.148.96
18) 199.59.148.229
19) 199.59.148.20
20) 157.240.17.35
21) 31.13.94.37
22) 31.13.68.169
23) 31.13.88.26
24) 31.13.88.169
25) 31.13.94.36
26) 31.13.94.49
27) 31.13.94.41
28) 31.13.94.10
29) 31.13.73.169
30) 31.13.73.9
31) 31.13.112.9
32) 31.13.106.4
33) 31.13.96.195
34) 31.13.82.169
35) 31.13.86.21
36) 31.13.85.169
37) 31.13.85.53
38) 31.13.96.194
39) 31.13.96.208
40) 31.13.96.193
41) 31.13.96.192
42) 31.13.112.4
43) 31.13.80.169
44) 31.13.95.169
45) 104.244.43.136
46) 199.96.61.1
47) 104.244.43.52
48) 104.244.43.6
49) 157.240.0.18
50) 104.244.43.231
51) 104.244.43.35
52) 66.220.148.145
53) 31.13.95.34
54) 31.13.95.48
55) 173.252.105.21
56) 31.13.95.35
57) 157.240.0.35
58) 173.252.108.21
59) 157.240.11.40
60) 104.244.43.167
61) 199.59.150.39
62) 199.59.149.230
63) 199.59.149.208
64) 199.16.158.104
65) 199.16.156.38
66) 96.44.137.28
67) 69.50.221.20
68) 69.30.25.21
69) 69.197.153.180
70) 69.162.134.178
71) 65.49.26.99
72) 65.49.26.98
73) 65.49.26.97
74) 50.23.209.199
75) 199.59.150.45
76) 199.59.150.44
77) 199.59.150.43
78) 199.59.150.40
79) 199.59.150.13
80) 199.59.150.12
81) 199.59.149.239
82) 199.59.149.238
83) 199.59.149.237
84) 199.59.149.236
85) 199.59.149.235
86) 199.59.149.234
87) 199.59.149.232
88) 199.59.149.231
89) 199.59.149.210
90) 199.59.149.207
91) 199.59.149.206
92) 199.59.149.205
93) 199.59.149.203
94) 199.59.149.202
95) 199.59.149.201
96) 199.59.148.9
97) 199.59.148.89
98) 199.59.148.8
99) 199.59.148.7
100) 199.59.148.6
101) 199.59.148.247
102) 199.59.148.246
103) 199.59.148.222
104) 199.59.148.206
105) 199.59.148.202
106) 199.59.148.201
107) 199.59.148.15
108) 199.59.148.147
109) 199.59.148.106
110) 199.59.148.102
111) 199.16.156.75
112) 199.16.156.71
113) 199.16.156.39
114) 199.16.156.11
115) 199.16.156.103
116) 184.72.1.148
117) 182.50.139.56
118) 173.255.213.90
119) 173.255.209.47
120) 173.252.248.244
121) 173.236.212.42
122) 173.236.182.137
123) 173.234.53.168
124) 173.231.12.107
125) 173.208.182.68
126) 168.143.171.93
127) 168.143.171.189
128) 168.143.171.186
129) 168.143.171.154
130) 168.143.162.58
131) 168.143.162.42
132) 128.242.250.157
133) 128.242.250.155
134) 128.242.250.148
135) 128.242.245.93
136) 128.242.245.43
137) 128.242.245.29
138) 128.242.245.253
139) 128.242.245.244
140) 128.242.245.221
141) 128.242.245.212
142) 128.242.245.189
143) 128.242.245.180
144) 128.242.245.157
145) 128.242.245.125
146) 128.242.240.93
147) 128.242.240.61
148) 128.242.240.29
149) 128.242.240.253
150) 128.242.240.244
151) 128.242.240.221
152) 128.242.240.218
153) 128.242.240.212
154) 128.242.240.189
155) 128.242.240.180
156) 128.242.240.157
157) 128.242.240.125
158) 128.121.243.77
159) 128.121.243.76
160) 128.121.243.75
161) 128.121.243.107
162) 128.121.243.106
163) 122.248.226.57
164) 159.106.121.75
165) 59.24.3.173
166) 66.220.146.94
167) 66.220.147.11
168) 66.220.149.18
169) 66.220.149.32
170) 69.171.224.40
171) 69.171.227.37
172) 69.171.228.74
173) 69.171.229.11
174) 69.171.229.73
175) 69.171.234.48
176) 69.171.242.11
177) 69.171.247.32
178) 69.171.247.71
179) 69.63.176.143
180) 69.63.176.15
181) 69.63.176.59
182) 69.63.178.13
183) 69.63.180.173
184) 69.63.181.12
185) 69.63.184.14
186) 69.63.184.142
187) 69.63.184.30
188) 69.63.186.30
189) 69.63.186.31
190) 69.63.187.12
191) 69.63.190.26
192) 67.15.100.252
193) 67.15.129.210
194) 199.16.156.40
195) 199.16.156.7
196) 199.16.158.190
197) 199.59.148.209
198) 199.59.148.97
199) 199.59.149.136
200) 199.59.149.244
201) 199.59.150.49
202) 88.191.249.182
203) 88.191.249.183
204) 208.101.21.43
205) 208.101.60.87
206) 208.43.170.231
207) 208.43.237.140
208) 67.228.102.32
209) 67.228.235.91
210) 67.228.235.93
211) 74.86.118.24
212) 74.86.12.172
213) 74.86.12.173
214) 74.86.142.55
215) 74.86.151.162
216) 74.86.151.167
217) 74.86.17.48
218) 74.86.226.234
219) 74.86.228.110
220) 74.86.3.208
221) 75.126.115.192
222) 75.126.124.162
223) 75.126.135.131
224) 75.126.150.210
225) 75.126.164.178
226) 75.126.33.156
227) 205.186.152.122
228) 64.13.192.74
229) 64.13.192.76
230) 104.16.252.55
231) 104.16.251.55
232) 210.56.51.193
233) 23.234.30.58
234) 210.56.51.192
235) 202.53.137.209
236) 156.233.67.243
237) 154.85.102.32
238) 154.92.16.97
239) 154.83.15.20
240) 154.85.102.30
241) 154.83.15.45
242) 154.83.14.134
243) 103.214.168.106
244) 150.107.3.176
245) 103.97.176.73
246) 103.73.161.52
247) 103.200.31.172
248) 52.175.9.80
249) 159.65.107.38
250) 59.188.250.54
251) 4.78.139.50
252) 93.179.102.140
253) 148.163.48.215
254) 54.89.135.129
255) 4.78.139.54
256) 23.101.24.70
257) 199.193.116.105
258) 162.220.12.226
259) 98.159.108.61
260) 50.87.93.246
261) 47.88.58.234
262) 98.159.108.58
263) 67.230.169.182
264) 159.138.20.20
265) 124.11.210.175
266) 98.159.108.57
267) 111.243.214.169
268) 103.42.176.244
269) 210.209.84.142
270) 107.181.166.244
271) 98.159.108.71
272) 23.225.141.210
273) 114.43.24.59
274) 45.77.186.255
275) 202.182.98.125
276) 45.114.11.25
277) 203.111.254.117
278) 103.240.180.117
279) 45.114.11.238
280) 43.226.16.8
281) 116.89.243.8
282) 80.87.199.46
283) 198.27.124.186
284) 39.109.122.128
285) 103.228.130.27
286) 118.107.180.216
287) 103.97.3.19
288) 103.56.16.112
289) 115.126.100.160
290) 118.193.202.219
291) 118.184.78.78
292) 118.193.240.41
293) 118.184.26.113
294) 103.39.76.66
295) 103.240.182.55
296) 103.230.123.190
297) 103.246.246.144
298) 103.200.30.143
299) 118.193.240.37
300) 198.44.185.131
301) 211.104.160.39
302) 103.228.130.61
303) 122.10.85.4
304) 103.226.246.99
305) 208.31.254.33
306) 50.117.117.42
307) 157.240.15.8
308) 157.240.13.8
309) 157.240.1.33
310) 157.240.12.5
311) 31.13.87.34
312) 31.13.91.33
313) 31.13.90.33
314) 31.13.87.33
315) 31.13.90.19
316) 31.13.95.18
317) 31.13.87.19
318) 31.13.83.34
319) 31.13.84.34
320) 31.13.85.34
321) 31.13.82.33
322) 31.13.76.99
323) 31.13.76.65
324) 31.13.75.12
325) 31.13.71.19
326) 31.13.70.33
327) 157.240.7.5
328) 31.13.70.13
329) 31.13.67.33
330) 157.240.7.8
331) 31.13.92.5
332) 31.13.91.6
333) 31.13.84.2
334) 31.13.87.9
335) 31.13.83.2
336) 31.13.85.2
337) 31.13.75.5
338) 31.13.70.9
339) 104.244.43.57
340) 199.59.149.204
341) 202.160.128.16
342) 199.96.58.15
343) 185.45.6.103
344) 202.160.128.203
345) 104.244.46.165
346) 199.96.63.177
347) 103.252.115.53
348) 104.244.46.17
349) 202.160.128.210
350) 104.244.46.211
351) 202.160.130.52
352) 104.244.46.5
353) 199.96.59.19
354) 104.244.46.85
355) 202.160.129.6
356) 103.252.114.101
357) 103.252.115.59
358) 103.252.115.49
359) 199.96.63.53
360) 192.133.77.197
361) 199.96.62.41
362) 104.244.43.234
363) 202.160.128.40
364) 202.160.128.96
365) 192.133.77.189
366) 202.160.128.14
367) 199.96.59.95
368) 104.244.43.104
369) 192.133.77.191
370) 192.133.77.145
371) 104.244.46.57
372) 185.45.7.185
373) 104.244.46.208
374) 104.244.46.186
375) 104.244.43.208
376) 103.252.114.11
377) 202.160.129.37
378) 199.96.62.75
379) 199.96.58.85
380) 104.244.46.185
381) 185.45.7.97
382) 104.244.46.93
383) 104.244.43.229
384) 104.244.43.182
385) 202.160.128.195
386) 185.45.6.57
387) 199.96.63.163
388) 199.96.58.177
389) 199.96.59.61
390) 199.96.63.75
391) 104.244.46.9
392) 202.160.128.238
393) 104.244.46.21
394) 103.252.115.153
395) 199.96.58.157
396) 192.133.77.59
397) 202.160.128.205
398) 199.96.62.21
399) 202.160.130.118
400) 104.244.43.228
401) 202.160.130.66
402) 185.45.7.189
403) 192.133.77.133
404) 185.45.7.165
405) 202.160.130.145
406) 104.244.43.128
407) 202.160.129.36
408) 103.252.115.221
409) 103.252.114.61
410) 199.96.58.105
411) 103.252.115.169
412) 104.244.46.246
413) 104.244.43.248
414) 104.244.46.63
415) 202.160.130.117
416) 104.244.46.52
417) 199.96.62.17
418) 202.160.129.164
419) 179.60.193.16
420) 157.240.3.50
421) 157.240.16.50
422) 157.240.2.50
423) 185.60.218.50
424) 162.125.83.1
425) 162.125.6.1
426) 162.125.8.1
427) 31.13.95.17
428) 157.240.17.14
429) 157.240.2.36
430) 185.60.216.36
431) 185.60.219.36
432) 157.240.2.14
433) 185.60.216.169
434) 157.240.8.50
435) 157.240.8.41
436) 157.240.3.8
437) 185.60.219.41
438) 157.240.12.50
439) 31.13.95.38
440) 31.13.81.4
441) 31.13.67.19
442) 157.240.6.35
443) 31.13.94.7
444) 157.240.20.18
445) 179.60.193.9
446) 31.13.94.23
447) 31.13.80.54
448) 31.13.69.169
449) 31.13.67.41
450) 31.13.64.7
451) 157.240.21.9
452) 157.240.20.8
453) 157.240.17.41
454) 157.240.18.18
455) 157.240.10.41
456) 157.240.1.50
457) 157.240.12.35
458) 157.240.1.9
459) 173.244.217.42
460) 173.244.209.150
461) 209.95.56.60
462) 31.13.95.37
463) 31.13.80.37
464) 157.240.8.36
465) 157.240.9.36
466) 157.240.17.36
467) 157.240.12.36
468) 157.240.10.36
469) 173.252.108.3
470) 31.13.69.245
471) 104.244.46.244
472) 104.244.45.246
473) 104.244.46.71
474) 162.125.32.6
475) 162.125.18.129
476) 162.125.80.5
477) 162.125.32.10
478) 162.125.17.131
479) 162.125.32.15
480) 162.125.2.6
481) 162.125.80.6
482) 162.125.80.3
483) 162.125.2.3
484) 162.125.34.133
485) 162.125.32.12
486) 162.125.1.8
487) 162.125.18.133
488) 162.125.32.2
489) 162.125.2.5
490) 162.125.32.5
491) 162.125.32.13
492) 162.125.32.9
493) 162.125.82.7
494) 162.125.7.1
495) 119.28.87.227
496) 31.13.95.33
497) 104.23.124.189
498) 104.23.125.189
499) 130.211.15.150
500) 104.31.142.88
501) 38.121.72.166
502) 65.49.68.152
503) 204.79.197.217
504) 54.234.18.200
505) 52.58.1.161
506) 184.173.136.86
507) 174.37.243.85
508) 69.171.224.36
509) 108.160.161.20
510) 108.160.161.83
511) 108.160.162.104
512) 108.160.162.102
513) 108.160.162.109
514) 108.160.162.115
515) 108.160.163.106
516) 108.160.162.98
517) 108.160.162.31
518) 108.160.163.102
519) 108.160.163.112
520) 108.160.163.108
521) 108.160.163.116
522) 108.160.163.117
523) 108.160.165.147
524) 108.160.165.11
525) 108.160.165.139
526) 108.160.165.141
527) 108.160.165.211
528) 108.160.165.173
529) 108.160.165.212
530) 108.160.165.189
531) 108.160.165.48
532) 108.160.165.53
533) 108.160.165.62
534) 108.160.165.8
535) 108.160.165.9
536) 108.160.166.137
537) 108.160.166.253
538) 108.160.166.61
539) 108.160.166.148
540) 108.160.166.42
541) 108.160.166.142
542) 108.160.167.147
543) 108.160.166.49
544) 108.160.166.57
545) 108.160.166.62
546) 108.160.166.9
547) 108.160.167.30
548) 108.160.167.156
549) 108.160.167.148
550) 108.160.167.165
551) 108.160.167.158
552) 108.160.169.171
553) 108.160.167.159
554) 108.160.167.167
555) 108.160.167.174
556) 108.160.169.178
557) 108.160.169.175
558) 108.160.169.174
559) 108.160.169.179
560) 108.160.169.181
561) 108.160.170.39
562) 108.160.170.33
563) 108.160.169.37
564) 108.160.169.185
565) 108.160.169.186
566) 108.160.169.46
567) 108.160.169.55
568) 108.160.169.54
569) 108.160.170.26
570) 108.160.170.41
571) 108.160.170.43
572) 108.160.170.44
573) 108.160.172.1
574) 108.160.170.52
575) 128.121.146.101
576) 108.160.172.200
577) 108.160.172.232
578) 108.160.172.208
579) 108.160.172.204
580) 108.160.173.207
581) 128.121.146.235
582) 128.121.243.228
583) 128.121.146.109
584) 128.121.146.228
585) 128.242.240.20
586) 128.242.240.59
587) 128.121.243.235
588) 128.242.240.117
589) 128.242.240.155
590) 128.242.240.149
591) 128.242.240.85
592) 128.242.240.91

Every entry in the IPv6 pool contains the pattern 'face:b00c', which is characteristic of genuine IP addresses of facebook.com. The GFW's use of well-known addresses of Internet companies has been noted in the past [8 §3.2], [9 §5.1].

1) 2a03:2880:f130:0083:face:b00c:0:25de
2) 2a03:2880:f12c:0083:face:b00c:0:25de
3) 2a03:2880:f12c:0183:face:b00c:0:25de
4) 2a03:2880:f127:0083:face:b00c:0:25de
5) 2a03:2880:f126:0083:face:b00c:0:25de
6) 2a03:2880:f129:0083:face:b00c:0:25de
7) 2a03:2880:f12a:0083:face:b00c:0:25de
8) 2a03:2880:f11f:0083:face:b00c:0:25de
9) 2a03:2880:f127:0283:face:b00c:0:25de
10) 2a03:2880:f11c:8083:face:b00c:0:25de
11) 2a03:2880:f11c:8183:face:b00c:0:25de
12) 2a03:2880:f11b:0083:face:b00c:0:25de
13) 2a03:2880:f11a:0083:face:b00c:0:25de
14) 2a03:2880:f10e:0083:face:b00c:0:25de
15) 2a03:2880:f117:0083:face:b00c:0:25de
16) 2a03:2880:f10c:0083:face:b00c:0:25de
17) 2a03:2880:f112:0083:face:b00c:0:25de
18) 2a03:2880:f111:0083:face:b00c:0:25de
19) 2a03:2880:f10f:0083:face:b00c:0:25de
20) 2a03:2880:f10d:0083:face:b00c:0:25de
21) 2a03:2880:f10d:0183:face:b00c:0:25de
22) 2a03:2880:f10c:0283:face:b00c:0:25de
23) 2a03:2880:f10a:0083:face:b00c:0:25de
24) 2a03:2880:f102:0183:face:b00c:0:25de
25) 2a03:2880:f107:0083:face:b00c:0:25de
26) 2a03:2880:f134:0083:face:b00c:0:25de
27) 2a03:2880:f136:0083:face:b00c:0:25de
28) 2a03:2880:f12d:0083:face:b00c:0:25de
29) 2a03:2880:f134:0183:face:b00c:0:25de
30) 2a03:2880:f131:0083:face:b00c:0:25de

APPENDIX B
REVERSE-ENGINEERED DNS PARSING AND INJECTION ALGORITHM

The below C code is our attempt to reverse-engineer the faulty DNS query processing algorithm that caused Wallbleed. It reproduces the behavior of the DNS injectors affected by Wallbleed in all important respects. If `PATCHED` is false, the code implements the Wallbleed v1 vulnerability; if true, the partially patched Wallbleed v2 (see Section III-C and Section VII).

Having observed a DNS query and copied it to memory, the injection device parses out the QNAME to decide whether the query is one to be censored, and prepares a response if so. There are several bugs, the most significant of which is a failure to bounds-check DNS name label lengths against the message size.

```
1  // Check if msg is a DNS query for a name that should be censored.
2  // If so, change msg into a response in place and return the length
3  // of the response. If not, return 0.
4  size_t response(unsigned char *msg, size_t msg_len)
5  {
6      if (msg_len < 12 || (msg[2] & 0x80) != 0)          No response if message is too short or not a query.
7          return 0;
8
9      char qname[126];                                   The dot-delimited, null-terminated representation of
10     size_t qname_i = 0;                                 QNAME will be stored in qname.
11     // QNAME parsing loop.
12     size_t msg_i = 12;                                 msg_ptr is meant to track msg_i and point just past
13     unsigned char *msg_ptr;                             QNAME at the end of the loop.
14     for (;;) {
15         size_t label_len = msg[msg_i++];               Bug: no check that msg_i is in bounds.
16         msg_ptr = &msg[msg_i];                         Sync msg_ptr with msg_i.
17
18         if (label_len == 0)                            Exit condition 1: an empty label.
19             break;
20         if (msg_i > msg_len)                           Exit condition 2: the label length prefix just parsed
21             break;                                      was outside the message bounds.
22
23 #if !PATCHED
24         if (qname_i + 1 > 124)                         Exit condition 3: not enough room for at least one
25             break;                                      byte of label (with a dot and a null terminator).
26 #else
27         msg_i += MIN(label_len, 124 - qname_i);        Take as much of the label as will fit, leaving room for
28                                                         a dot and a null terminator.
29         if (qname_i + label_len > 124)                 Exit condition 4: label too long to fit in qname.
30             break;                                      Bug: msg_ptr ≠ msg + msg_i in this case.
31 #endif
32
33         if (qname_i > 0)                               Append a dot to qname, if not the first label.
34             qname[qname_i++] = '.';
35         size_t n = MIN(label_len, 125 - qname_i);      Copy as much of the label as will fit in qname.
36         memcpy(qname + qname_i, msg_ptr, n);            Bug: no check that msg_ptr + n is in bounds.
37         qname_i += n;
38 #if !PATCHED
39         msg_i += n;
40         if (n < label_len)                             Exit condition 4: label too long to fit in qname.
41             break;                                      Bug: qname_ptr ≠ msg_ptr in this case.
42 #endif
43     }
44     qname[qname_i] = '\0';                             Null-terminate the dot-delimited name string.
45
46     if (!name_matches_blocklist(qname))                Does the extracted QNAME string match the
47         return 0;                                       blocklist? If not, do not send a response.
48
49     // Read QTYPE.
50     uint16_t qtype  = ntohs(*(uint16_t *) msg_ptr);    msg_ptr may not agree with msg_i here.
51 #if PATCHED
52     // Read QCLASS, enforce QCLASS == 1.
53     uint16_t qclass = ntohs(*(uint16_t *) (msg_ptr + 2));
54     if (qclass != 1)
55         return 0;
56 #endif
```

```
57      // Change the query into a response.
58      size_t resp_len = msg_i + 4;                    Add 4 bytes for the query's QTYPE and QCLASS.
59      if ((msg[2] & 0x01) == 0)                       If the RD flag was not set in the query, set AD in the
60          *(uint16_t *) &msg[2] = htons(0x8400);       response.
61      else                                            If RD was set in the query, set RD and RA in the
62          *(uint16_t *) &msg[2] = htons(0x8180);       response.
63      *(uint16_t *) &msg[ 4] = htons(1);              QDCOUNT = 1
64      *(uint16_t *) &msg[ 6] = htons(1);              ANCOUNT = 1
65      *(uint16_t *) &msg[ 8] = htons(0);              NSCOUNT = 0
66      *(uint16_t *) &msg[10] = htons(0);              ARCOUNT = 0
67      // Append an answer section according to QTYPE.
68      const unsigned char *rdata;
69      uint32_t rdlength;
70      if (qtype == 28) {                              Type AAAA queries get a type AAAA response.
71          rdlength = 16;
72          rdata = next_aaaa_address();                Get the next IPv6 address from the cyclic pool.
73      } else {                                        All other QTYPEs get a type A response.
74          qtype = 1;
75          rdlength = 4;
76          rdata = next_a_address();                   Get the next IPv4 address from the cyclic pool.
77      }
78      uint32_t ttl = rand_in_range(64, 254);          TTL is random between 64 and 254 inclusive.
79      unsigned char rr[] = {                          Construct a resource record.
80          0xc0, 0x0c, // NAME                        Compression pointer pointing back to QNAME.
81          0, 0,       // TYPE placeholder
82          0x00, 0x01, // CLASS = IN
83          0, 0, 0, 0, // TTL placeholder
84          0, 0,       // RDLENGTH placeholder
85      };
86      *(uint16_t *) &rr[ 2] = htons(qtype);           Set QTYPE.
87      *(uint32_t *) &rr[ 6] = htonl(ttl);             Set TTL.
88      *(uint16_t *) &rr[10] = htons(rdlength);        Set RDLENGTH.
89      memcpy(msg + resp_len, rr, sizeof(rr));         Append the resource record up to RDATA.
90      resp_len += sizeof(rr);
91      memcpy(msg + resp_len, rdata, rdlength);        Append the RDATA (the false IP address).
92      resp_len += rdlength;
93
94      return resp_len;                                This query gets an injected response.
95  }
```

We do not know exactly how a packet payload arrives in memory after being observed. It may be, for example, a software copy, or an automatic DMA transfer from the network interface. However it happens, a few minor peculiarities of the DNS injectors' behavior are best explained as artifacts of how packets are copied into memory. These are: that the 18th byte of the memory buffer is always zero (footnote 3); that at one time, the first 4 bytes of leaked memory were different from the others ("digest" bytes, Section IV-B); and that the `msg_len` limit in the `response` function above comes from the UDP header, not the number of bytes actually available in the packet (when those quantities differ).

```
1   // Copy an observed UDP packet payload to memory for analysis and
2   // possible modification, and inject a response if it is a DNS
3   // query for a name on the blocklist. hdr_len is the size of the
4   // payload in the UDP header (not counting the header itself).
5   void udp_packet_callback(const void *data, size_t data_len, size_t hdr_len)
6   {
7       data_len = MIN(data_len, hdr_len);          Trim packet payload to the size in the header.
8       void *work = allocate_memory(hdr_len + 28); Memory for the query and a response record.
9       memset(work, 0x00, 18);                     Clear the beginning of the buffer; see footnote 3.
10      memcpy(work, data, data_len);               Copy the packet to working memory. The copy uses
11                                                   data_len, but parsing uses hdr_len.
12      if (USE_DIGEST)                             "Digest" bytes (Section IV-B), when present, are just
13          memset(work + data_len, 'D', 4);         after the query. We are using a fixed byte pattern.
14      size_t resp_len = response(work, hdr_len);  Is this packet a DNS query that needs a response?
15      if (resp_len > 0)                           If so, inject the response. We are omitting details of
16          inject(work, resp_len);                  spoofing the source address, etc.
17      free_memory(work);
18  }
```