L↻↻py Hell(ow): Infinite Traffic Loops at the Application Layer

Yepeng Pan
CISPA Helmholtz Center for Information Security

Anna Ascheman
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

Denial-of-Service (DoS) attacks have long been a persistent threat to network infrastructures. Existing attack primitives require attackers to *continuously* send traffic, such as in SYN floods, amplification attacks, or application-layer DoS. In contrast, we study the threat of application-layer traffic loops, which are an almost cost-free attack primitive alternative. Such loops exist, e.g., if two servers consider messages sent to each other as malformed and respond with errors that again trigger error messages. Attackers can send a *single* IP-spoofed loop trigger packet to initiate an *infinite* loop among two servers. But despite the severity of traffic loops, to the best of our knowledge, they have never been studied in greater detail.

In this paper, we thus investigate the threat of application-layer traffic loops. To this end, we propose a systematic approach to identify loops among real servers. Our core idea is to learn the response functions of all servers of a given application-layer protocol, encode this knowledge into a loop graph, and finally, traverse the graph to spot looping server pairs. Using the proposed method, we examined traffic loops among servers running both popular (DNS, NTP, and TFTP) and legacy (Daytime, Time, Active Users, Chargen, QOTD, and Echo) UDP protocols and confirmed the prevalence of traffic loops. In total, we identified approximately 296k servers in IPv4 vulnerable to traffic loops, providing attackers the opportunity to abuse billions of loop pairs.

1 Introduction

Traffic loops *at the network layer* are a widely-known occasional configuration issue that attackers can abuse to launch severe Denial-of-Service (DoS) attacks that overload networks and hosts. Such routing loops typically occur due to accidental router misconfigurations. Fortunately, the pioneers of the internet realized the problem of network loops early on and stopped it by design. When standardizing the Internet Protocol (IP), they included a dedicated IP header field to prevent packets from looping indefinitely. Any IP router decrements the value of this *Time-to-Live* (TTL) header field when it forwards a packet to the next hop. By discarding packets with TTL=0, routers enforce that IP packets do not spin infinite cycles — mitigating loops at the network layer.

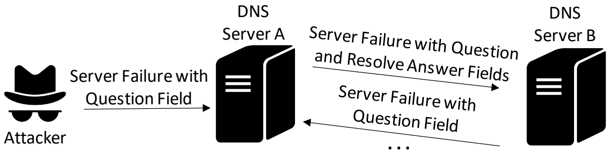

DNS
Server A
Server Failure with Question and Resolve Answer Fields
DNS
Server B
Server Failure with Question Field
Attacker
Server Failure with Question Field
. . .

Figure 1: Example DNS traffic loop. The attacker (left) sends a single IP-spoofed, malformed DNS message to a server (middle), impersonating the IP address owner (right). This loop-trigger message combines a server failure with a valid question field, which server A resolves and replies to B. Server B then crafts a payload that preserves the server failure opcode and question field but strips the answer field. To this point, a packet loop is formed between the two servers.

In this paper, we show that traffic loops also exist *at the application layer*, irrespective of network-layer defenses. Surprisingly, app-layer loops have gained little attention except for an isolated past vulnerability in NTP in 2009 [38] or reports concerning only two legacy protocols [16]. Application-layer loops arise if two network services indefinitely respond to each other's application-layer messages. Such loops remain unnoticed by IP's TTL mechanism and are hence infinite. Without losing generality, imagine two DNS resolvers that respond with an error message when receiving an error message as input. If an error as input creates an error as output, and a second system behaves the same, these two systems will keep sending error messages back and forth — *indefinitely*. Attackers can invoke such endless loops by sending a single IP-spoofed packet. Worse, they can spawn loops in parallel. Figure 1 shows an example loop we identified among real DNS servers. In this example, an attacker can cause a loop among two faulty DNS servers by injecting a single DNS server failure message. Once injected, the servers send DNS error messages back and forth — indefinitely.

Application-layer loops are an overwhelming attack primitive that can be abused for, e.g., launching link-flooding DoS attacks, overloading networks that host looping servers, exhausting computation resources of looping servers themselves, or economically damaging ASes due to increased upstream costs. Worse, attackers do not require large attack bandwidth to abuse loops. In fact, sending a *single* IP-spoofed packet creates an *infinite* loop among two servers. Moreover, attackers do not require any control of the looping servers or their configurations; all it requires is finding pairs of servers that loop. Application-layer loops thus provide the attackers with several benefits over existing DoS attack primitives, where attackers either have to *continuously* send traffic (e.g., SYN floods [17], amplification DDoS [8, 10, 14, 22, 24, 27]) and/or have to *control configurations* of the abused infrastructure (e.g., DNS-based attack chains that require partial control of the DNS zones [15, 36]). We thus believe that it is of utmost importance to systematically study the overall threat.

To this end, we devise a scalable methodology to identify application-layer loop internet infrastructures. We bootstrap our loop search with a list of known services of a given UDP-based application-layer protocol from past network scans. We then send a few hand-crafted discovery probes to these known services. These protocol-specific probes reflect protocol messages for which we do not expect answers, such as protocol-specific error messages or semantically invalid requests and responses. Seeing that many services unexpectedly *do* respond, we cluster their responses to abstract away syntactical differences while retaining semantic similarities. Next, we learn the *response function* per server, allowing us to reliably predict which response a server would send after receiving a given input. We then use these mappings to form a *loop graph* that encodes which response a server returns when receiving a given response as input. We traverse the loop graph to find potential loop pairs, i.e., two hosts that will send messages forth and back indefinitely. Finally, we randomly sample from the potential loop pairs and use a rate-limited proxy to verify their loop behavior in practice.

We apply this methodology to study loops in three popular UDP-based application-layer protocols (DNS, NTP, and TFTP) and six legacy protocols for which we suspect loop issues by design (Chargen, QOTD, Echo, Time, Daytime, and Active Users). We find that loops at the application layer are more common than intuitively expected. In fact, they seem to be a widespread problem of server implementations for all of the studied protocols. In DNS, NTP, and TFTP alone, we find 249K "loop hosts," i.e., hosts that attackers can abuse to construct a loop when being paired with other loop hosts. We empirically validate that this corresponds to over 5.3B potential "loop host pairs" of which we can probabilistically confirm 87.9%, affecting 7,318 ASes (9.7%) globally. In addition, we find 123.6k servers running legacy protocols, corresponding to 2.0B potential "loop host pairs," and we can probabilistically confirm that 78.1% of them can be abused. Unfortunately, we empirically validate that loops even exist *across protocols*, rendering them extremely hard to mitigate on a per-protocol basis alone.

To better understand mitigation options, we study the reasons for traffic loops. First and foremost, we find that responding to error messages or random data can quickly turn into looping behaviors. Second, some loops are protocol-specific, such as middlebox-induced loops in DNS, where servers react to well-formed DNS responses with another DNS response. Finally, legacy protocols such as Chargen and QOTD send responses irrespective of the payload they receive, entirely lacking a notion of requests and responses. This shows that standardization bodies and developers must be more careful about traffic loops. We were surprised to see that few standards explicitly mitigate traffic loops. Even *if* standards implicitly describe patterns of how to avoid loops (e.g., "do not reply to error messages when receiving an error message"), implementations either do not follow such optional advice, or the standard is not explicit enough (e.g., it does not specify how to react upon invalid messages such as random data). To increase awareness of this threat, we performed a large-scale disclosure operation, reaching out to the standardization experts and developers of the affected implementations.

2 Background and Related Work

Traffic Loops Traffic loops are a strong primitive for performing DoS attacks. Known classes of loops either operate at the network layer or are application-layer attacks that require (partial) control over service configurations.

Nosyk et al. found that *network-layer routing loops* can be used as mega amplifiers for DNS-based DoS attacks [37]. In such a routing loop, which is mostly caused by misconfigurations and inconsistent implementations of routing protocols, a packet circulates through a sequence of routers until its TTL decays. Given that a DNS censorship middlebox injects a DNS response upon observing a censored DNS query [39], a single DNS query can trigger repetitive answers from the middlebox if the middlebox is involved in routing loops. This way, an attacker can use routing loops to generate and send a huge amount of traffic to a target victim. Plenty of research [21, 30, 41] focuses on effectively identifying and eliminating routing loops.

However, there is little research on *application-layer loops*, and existing research is tailored to certain protocols and requires partial attacker control over configurations. For example, TsuNAME [36] is an app-layer loop attack against DNS servers. The attack exploits cyclic dependencies in DNS zone files. Two authoritative DNS resolvers with misconfigured zone files can refer to each other for resolving, and thus, a triggering query from the attacker can start a traffic loop between the two servers. Another application-layer loop attack abuses CNAME records [15]. By manipulating the CNAME records on two authoritative name servers, an attacker can chain the CNAME records on two name servers to form an aliasing loop. However, to leverage these two attacks, attackers need to set up desired zone configurations or domain records on target servers. In [18], Chen et al. introduced a DoS attack against Content Delivery Networks (CDNs), where malicious customers can create forwarding loops among or inside CDNs by manipulating the forwarding process.

Existing traffic loops thus either rely on temporal network configuration errors or are tailored to certain application-layer protocols and require a high degree of attacker control. In contrast, the loops that we identify do not require any privileges or network errors and generalize to several application-layer protocols.

Related DoS Attack Primitives Once found, the traffic loops that we present in the paper can be abused as an almost infinitely-amplifying attack primitive (offering more severe amplification than previously known amplification attacks) and assist attackers in performing link-flooding attacks without requiring control over the endpoints involved in the attack. We discuss both primitives in the following.

Amplification attacks exploit network services that operate application-layer protocols with known amplification vectors [11, 33, 40]. For example, by spoofing the IP address of victim hosts and sending floods of query packets to DNS servers, an attacker can generate and direct significant attack traffic in the form of DNS responses to the victim. In amplification attacks, an attacker needs to continuously send floods of IP-spoofed packets to maintain the attack traffic — in contrast to needing only a single trigger message in traffic loops. Other than DNS [24], protocols such as NTP [27], SSDP [10], BitTorrent [8], and SNMP [22] can also be abused in such attacks. Knowing the threats of amplification attacks, the security community has also discussed mitigations. Source address validation (SAV) [28] is considered the best practice, as amplification attacks require an attacker to send IP-spoofed packets. However, a 2019 study [34] showed that over 21% of analyzed ASes do not deploy such outbound filtering.

Link-flooding attacks aim to overload entire network links, not just single hosts. The Coremelt attack [43] is an example in which thousands of attacker-controlled bots exchange low amounts of traffic to overload a network link they share. Similarly, the Crossfire attack commands bots to send traffic to public servers instead of forming pairs of bots [26]. But to coordinate either link-flooding attack, attackers require bots, i.e., compromised hosts. In contrast, loops abuse existing, non-compromised servers, which readily provide billions of loopable server pairs. An attacker can selectively pick loopable servers on victim links and trigger application-layer loops between them to launch a link-flooding attack. Such attacks can only be mitigated by SDN-based mitigations [25, 32, 44, 46].

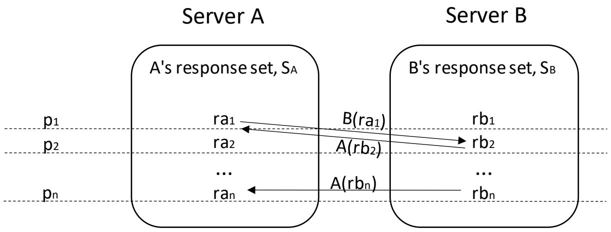


Figure 2: Example response functions of servers.

3 Methodology

Before discussing the details of our methodology, we first present the definition of a loop pair and formulate the research problem of finding loop pairs. Next, to show the underlying challenges, we discuss the limitations of a naïve straw-man approach. Finally, we introduce our methodology in detail.

3.1 Loop DoS Problem Formulation

Consider two UDP servers A and B. Upon receiving a packet p_i, A and B deterministically respond to a packet p_i using response functions $A(p_i) = ra_i$ and $B(p_i) = rb_i$, respectively. The response functions map a vector of incoming packets $P = p_1, p_2, \cdots, p_n$ to server-specific response sets, i.e., $S_A = \{(p_1, ra_1), (p_2, ra_2), \cdots, (p_n, ra_n)\}$ for server A and $S_B = \{(p_1, rb_1), (p_2, rb_2), \cdots, (p_n, rb_n)\}$ for B. Figure 2 illustrates this mapping. The mapping can now be used to find loops at the application layer. In the most simple case, when $A(rb_i) = ra_m$ and $B(ra_m) = rb_i$, we have $B(A(rb_i)) = rb_i$. That is, whenever A sends ra_m to B, B will respond with rb_i to A, which in turn sends ra_m, and so on — an endless loop. The attacker can trigger this loop by sending an IP-spoofed rb_i packet to A, forging the identity of B as the packet source (or alternatively: send rb_m to B in the name of A).

But loops may also exist in more complex settings that involve multiple response conversions. To generalize, when there are $B(\cdots A(B(A(rb_i)))) = rb_i$ (or $A(\cdots B(A(B(ra_i)))) = ra_i$), a loop between server A and B is possible and we call server A and server B a *loop pair*. This generalization allows us to find loop pairs for which $B(A(rb_i)) \neq rb_i$. For example, consider $A(rb_i) = ra_j$, $B(ra_j) = rb_k$, $A(rb_k) = ra_l$, and $B(ra_l) = rb_i$ — completing a loop after four response types.

In the following sections, we outline methodologies for how to identify possible loop pairs among servers. Basically, we need to learn the servers' response sets (e.g., S_A) or their response function (e.g., $A(p)$). The difficulty lies in the fact that the number of server pairs is quite large and the input space infinite — thus calling for scalable methods.

3.2 Straw-Man Approach for Finding Loops

Given two UDP servers, A and B, a naïve approach to identify possible loop pairs could operate as follows:

1. To learn the response sets (S_A and S_B) of servers A and B, we could generate random packets and collect A's and B's responses to these generated packets respectively.

2. We could then send all packets $ra_i \in S_A$ to server B and all packets $rb_m \in S_B$ to server A to learn the response function $A(p)$ where $p \in \{rb_1, rb_2, \cdots, rb_n\}$ and $B(p)$ where $p \in \{ra_1, ra_2, \cdots, ra_n\}$.

3. If we find $B(ra_i) = rb_m$ and $A(rb_m) = ra_i$, we could assume that servers A and B form a loop pair.

However, this straw-man solution suffers from three substantial issues that underline some of the core challenges of discovering loop pairs in a scalable and precise manner.

First, the methodology does not scale. Learning all possible packets in a real server's response set requires us to send a large number of random probes to real servers. Identifying loops among servers by sending all packets in each server's response set to all servers again (step (2)) induces an impractical number of probes when searching for loop pairs among millions of servers. With n servers and m probe packets, the straw-man approach may require us to send up to $n * m$ probes *to each server*, quickly demonstrating that this does not scale for n in the range of millions and large m due to the random nature of the naïve packet generation.

Second, the automated and random nature of the packet exploration phase (step (1)) — often the basis for software vulnerability search [9] — may trigger undesired bugs, crash, or otherwise interfere with the underlying server software. As we are neither interested in triggering such bugs nor find it ethical to explore them in live systems, we have to resort to less intrusive ways of exploring the response functions.

Third, the response matching phase (step (3)) may miss loops as it searches for exact payload matches only. However, in many network protocols, two non-identical packets may share their semantic meaning and thus what they trigger. For example, consider two DNS responses that contain different IP addresses in two otherwise-identical responses (e.g., due to load balancing). If the response behavior is not influenced by the concrete IP addresses in the input packet but rather based on the type of the DNS message and the number of DNS resource records, the naïve search would falsely neglect a loop. More formally, consider a case where $B(ra_i) = rb_m$ and $A(rb_m) = ra'_i$, where $ra_i \approx ra'_i$ but $ra_i \neq ra'_i$. If ra'_i is semantically equivalent to ra_i and thus triggers the same behavior at B, it is a viable loop trigger but would have been dismissed by the naïve search that searches for strict equality.

3.3 Methodology Sketch

Considering the constraints of the straw-man approach, we now provide a methodology to identify loop pairs that tackles the aforementioned challenges. For the sake of concreteness, without losing generality, we describe our methodology with the help of a running example of finding loops among DNS resolvers. But our chosen methodology generalizes also to other UDP-based application-layer protocols. Figure 3 illustrates our five-step methodology when applied to DNS. In general, given a list of IP addresses of a given UDP protocol, we conduct the following five steps to find loops:

1. Discovery Probes (Section 3.4): For a given UDP application-layer protocol, we send hand-crafted discovery probe packets to all servers to learn their responses. The collected responses represent potential entry gates to loops. With reference to our running example (Figure 3), we choose to send two discovery probes: a non-empty, valid DNS response with a type `A` record and a DNS format error response. They trigger six responses from servers A–D.

2. Response Clustering (Section 3.5): We then group the responses according to their semantics, abstracting away potential syntactical differences. Each cluster (ideally) represents an equivalence class of the semantics of the responses in the cluster. In our running example, the six syntactically different responses can be grouped into four semantic clusters.

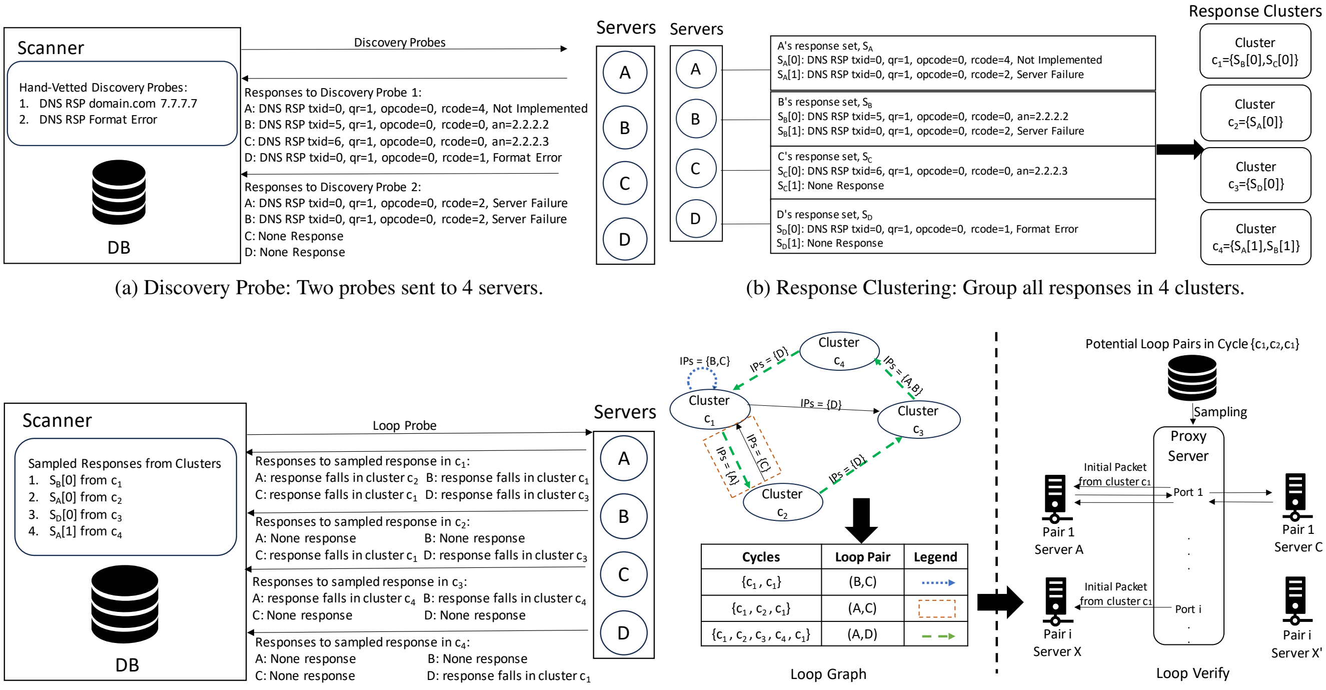
3. Loop Probes (Section 3.6): Next, we sample loop probe packets from the identified clusters and send them to all servers. That is, we choose random representatives from each semantic equivalence class and learn how all servers react to them. The responses to these loop probes are clustered in the same way as in the previous step. In our running example, we randomly pick one probe from each cluster. We then send all these sampled probes to A–D and receive one to four responses from each server.

4. Loop Graph (Section 3.7): After learning the server response sets, we build a loop graph that shows which servers (edges) reply with which response when receiving an input (nodes). We then search for cycles in this loop graph, which hint at loop pairs. In our DNS example, we can immediately spot a trivial self-loop for cluster c_1 among hosts B and C (in blue, dotted). But the loop graph also reveals more complex cycles. For example, hosts A and D form a cycle using the response sequence $\{c_1, c_2, c_3, c_4, c_1\}$, i.e., $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4 \rightarrow c_1$ (in green, dashed).

5. Loop Verification (Section 3.8): Finally, we verify loop pairs, e.g., (A,C), identified in the loop graph via a proxy server. In our DNS example, to verify loop pairs identified in the cycle $\{c_1, c_2, c_1\}$, the proxy stands as a person-in-the-middle between server A and server C. Then, the proxy uses its own IP to send a packet $S_B[0] \in c_1$ (see Figure 3 (c)) to host A and passively delegates further packets between the two hosts. If the proxy observes sufficient traffic between the two hosts, host A and host C are confirmed as a loop pair.

3.4 Discovery Probes

The first step in discovering loops is to learn the response set of real servers. To this end, we carefully craft hand-vetted discovery probes. In order to trigger diverse responses, we design


Scanner
Hand-Vetted Discovery Probes:
1. DNS RSP domain.com 7.7.7.7
2. DNS RSP Format Error
DB
Discovery Probes
Responses to Discovery Probe 1:
A: DNS RSP txid=0, qr=1, opcode=0, rcode=4, Not Implemented
B: DNS RSP txid=5, qr=1, opcode=0, rcode=0, an=2.2.2.2
C: DNS RSP txid=6, qr=1, opcode=0, rcode=0, an=2.2.2.3
D: DNS RSP txid=0, qr=1, opcode=0, rcode=1, Format Error
Responses to Discovery Probe 2:
A: DNS RSP txid=0, qr=1, opcode=0, rcode=2, Server Failure
B: DNS RSP txid=0, qr=1, opcode=0, rcode=2, Server Failure
C: None Response
D: None Response
Servers
A
B
C
D
Servers
A
B
C
D
A's response set, S_A
$S_A[0]$: DNS RSP txid=0, qr=1, opcode=0, rcode=4, Not Implemented
$S_A[1]$: DNS RSP txid=0, qr=1, opcode=0, rcode=2, Server Failure
B's response set, S_B
$S_B[0]$: DNS RSP txid=5, qr=1, opcode=0, rcode=0, an=2.2.2.2
$S_B[1]$: DNS RSP txid=0, qr=1, opcode=0, rcode=2, Server Failure
C's response set, S_C
$S_C[0]$: DNS RSP txid=6, qr=1, opcode=0, rcode=0, an=2.2.2.3
$S_C[1]$: None Response
D's response set, S_D
$S_D[0]$: DNS RSP txid=0, qr=1, opcode=0, rcode=1, Format Error
$S_D[1]$: None Response
Response Clusters
Cluster c_1={$S_B[0]$,$S_C[0]$}
Cluster c_2={$S_A[0]$}
Cluster c_3={$S_D[0]$}
Cluster c_4={$S_A[1]$,$S_B[1]$}
Scanner
Sampled Responses from Clusters
1. $S_B[0]$ from c_1
2. $S_A[0]$ from c_2
3. $S_D[0]$ from c_3
4. $S_A[1]$ from c_4
DB
Loop Probe
Responses to sampled response in c_1:
A: response falls in cluster c_2 B: response falls in cluster c_1
C: response falls in cluster c_1 D: response falls in cluster c_3
Responses to sampled response in c_2:
A: None response B: None response
C: response falls in cluster c_1 D: response falls in cluster c_3
Responses to sampled response in c_3:
A: response falls in cluster c_4 B: response falls in cluster c_4
C: None response D: None response
Responses to sampled response in c_4:
A: None response B: None response
C: None response D: response falls in cluster c_1
Servers
A
B
C
D
IPs = {B,C}
Cluster c_1
Cluster c_2
Cluster c_3
Cluster c_4
IPs = {D}
IPs = {A,B}
IPs = {D}
IPs = {A}
IPs = {C}
IPs = {D}
Loop Graph
Potential Loop Pairs in Cycle {c1,c2,c1}
Sampling
Proxy Server
Initial Packet from cluster c1
Port 1
Pair 1 Server A
Pair 1 Server C
Initial Packet from cluster c1
Port i
Pair i Server X
Pair i Server X'
Loop Verify

Cycles	Loop Pair	Legend
{c_1, c_1}	(B,C)	⋯▸
{c_1, c_2, c_1}	(A,C)	▭
{c_1, c_2, c_3, c_4, c_1}	(A,D)	– –▸

(a) Discovery Probe: Two probes sent to 4 servers.

(b) Response Clustering: Group all responses in 4 clusters.

(c) Loop Probe: Confront each server with samples from each cluster.

(d) Loop Graph & Loop Verify: Derive (left) and test (right) loop pairs.

Figure 3: Example of our 5-step methodology to identify loop pairs (here, e.g., (B,C)) among 4 DNS servers (A,B,C,D).

semantically different discovery probes according to various protocol aspects we find in their standards (e.g., RFCs). Figure 3 (a) provides a concrete example of discovery probes. For all four DNS servers, we send hand-vetted discovery probes, including DNS responses and DNS error messages. We will outline the concrete discovery probes in Section 4.1. After having crafted discovery probes for each of the analyzed application-layer protocols, we then use an off-the-shelf scanner to probe all servers of a given application-layer protocol. We save the responses for further clustering (Section 3.5).

The design choice to manually create discovery probes has pros and cons. Hand-vetted packets are inherently incomplete and cannot trigger all possible responses. This implies that we may underestimate the number of loop pairs we can find. We try to mitigate this challenge by creating probes with large semantic diversity, assisted by careful studies of the protocol specifications. Then again, in our eyes, hand-vetted probes are the most straightforward option to perform a large-scale experiment in an ethical manner. For each protocol, we only have around 30 hand-vetted discovery probes, which reduces the load to each probed system and limits the potential side effects on server stability during testing.

3.5 Response Clustering

In principle, each of the probed servers could respond to each discovery probe with a unique packet, resulting in millions of unique responses. As described in Section 3.2, to find potential loops, we need to send responses collected in Section 3.4 to all servers to learn servers' response functions. Sending this number of responses to each server is not scalable. We mitigate this scalability problem with an important observation. Namely, groups of the millions of unique responses are *semantically* the same; they are only *syntactically* different. For example, if two DNS responses only differ by the transaction ID header field, they should have the same effect. Grouping semantically-equivalent responses reduces the loop search to sending only a (few) representative(s) per group.

Thus, we cluster responses by abstracting away syntactic differences. For each UDP protocol, we identify important packet header fields and differentiate packets according to these critical fields. Each response falls exactly in one response cluster. Such a response cluster $c_i \in C$ is defined by several packet field values, where all packets in that response cluster exhibit the same values. The detailed clustering and critical field selection for each protocol is presented in Section 4. As shown in Figure 3 (b), when clustering DNS responses, we group packets according to the flags field and ignore syntactic differences such as TXID and IP addresses.

While clustering captures the behavior of most servers well, we also observe certain response behaviors that are not common among many probed hosts. As we are mainly interested in larger loops, we only consider response clusters with more than 10k unique hosts and ignore smaller ones.

3.6 Loop Probe

The clustered responses now help us to identify the servers' response functions. That is, for each server S, we would like to learn its mapping $S(c_i) = rs_i$ for all $c_i \in C$. Note that the response function operates on packets, and not on clusters. Then again, packets within a cluster share the same semantics. We thus sample five responses from each response cluster and use them as probes for all servers, which approximates the response function for *each cluster* instead of each packet.

We choose multiple instead of just one representative per cluster to verify the clustering step. In principle, samples from the same response cluster *shall* have the same effect. To verify this, we measure how many servers respond to how many probes of a given response cluster. If most servers respond to all five samples, it suggests the clustering performs well, and all samples indeed induce the same effect. Otherwise, we adjust the clustering discriminators (e.g., packet fields, see Section 3.5) and repeat the clustering and loop probing.

Note that the proposed approach only provides basic clustering validation. Real servers may generate semantically different responses to all five samples, which can thus undermine our validation. To improve the reliability of the verification approach, one can further check whether most servers generate semantically same responses for five samples.

At the end of the loop probe phase, we use the same response clustering methodology to group all responses to the sampled loop probes. By doing this, we can approximate each server's response function for the known response clusters.

In the example from Figure 3 (c), we use sampled responses from each response cluster to scan all servers again. After the clustering of the scan result, we learn that when receiving a packet from cluster c_1, DNS server A sends a packet in cluster c_2, i.e., server A has response function $A(c_1) = c_2$.

3.7 Loop Graph and Loop Search

Now, we have gathered all essential ingredients — the servers' response sets (Section 3.4) and response functions (Section 3.6) — to identify loop pairs among the probed servers. We encode this information in a directed graph where nodes are response clusters c_i and each edge $e_i = [c_i, c_m]$ is the set of server IPs with response function $F(c_i) = c_m$. Then, we conduct a depth-limited search to find possible cycles in the graph. Each cycle indicates possible loops among servers. Intuitively, cycles correspond to paths that span one or multiple response clusters between two alternating hosts.

Shorter cycles are easy to spot in the loop graph. When the cycle is of length one — i.e., a single response payload forms the loop — every IP involved in the edge can be paired with other IPs on the same edge to form a loop pair. Consider the example cycle $\{c_1, c_1\}$ shown in Figure 3 (d). Servers B and C on edge $[c_1, c_1]$ form a loop pair, and an attacker can trigger the loop by sending an IP-spoofed packet in cluster c_1 from server B to server C. For cycles of length two, each IP on one edge can form a loop pair with each IP on the other edge. Consider the example cycle $\{c_1, c_2, c_1\}$ in Figure 3 (d). By sending an IP-spoofed packet in cluster c_1 from server C to server A, an attacker can trigger a loop between server A and server C, where $A \in [c_1, c_2]$ and $C \in [c_2, c_1]$.

However, significantly longer cycles are also possible. We can generalize our cycle search to arbitrary cycle lengths. A cycle of length n from server A to B exists if there is a path of n edges alternating between A and B. We index edges involved in such a cycle as $e_1, e_2, \cdots, e_n$. Server A where $A \in e_1 \cap e_3 \cap \cdots \cap e_{n-1}$ ("odd edge indexes") and server B where $B \in e_2 \cap e_4 \cap \cdots \cap e_n$ ("even edge indexes") form a loop pair of cycle length n. Consider the example cycle $\{c_1, c_2, c_3, c_4, c_1\}$ with $n = 4$ in Figure 3 (d). Here, server A where $A \in [c_1, c_2] \cap [c_3, c_4]$ forms a loop pair with server D where $D \in [c_2, c_3] \cap [c_4, c_1]$. An attacker can trigger a loop by sending an IP-spoofed packet, e.g., of cluster c_1 or c_3 from server D to server A.

To find all potential loop pairs of a given protocol, we perform an exhaustive path search up to cycle length n over the protocol's loop graph. For the protocols we studied, loop graphs span 46–3,778 nodes. The complete search up to $n = 4$ terminates within 10 seconds (see Section 4). Our methodology is thus systematic, scalable, and still reasonably complete in identifying loop pairs for a given protocol.

3.8 Loop Verify

Finally, we verify the potential loop pairs to achieve two goals. First, the potential loop pairs identified in the loop graph are not necessarily vulnerable to abuse. For example, servers could aggressively limit the number of requests per IP address. In this case, a loop will stop automatically after reaching the rate limit. Such defenses cannot be identified by low-rate scans. Second, a final end-to-end attack verification tests if our methodology for identifying loops is reasonable.

We thus propose a proxy to verify loop pairs probabilistically. Figure 3 (d) illustrates the overall setup of the loop verify proxy. For each cycle in the loop graph, we sample a subset of loop pairs. The proxy serves as a person-in-the-middle between each sampled loop pair. This way, we can control the speed of the loop and stop it at any point, allowing for tests that do not overload real systems. Next, for loop pairs sampled from a certain cycle, e.g., $\{c1, c2, c1\}$, the proxy chooses a packet from cluster c_1 to initiate a loop. After the initial packet, the proxy server passively waits and delegates any packets between servers. Once the proxy observes sufficiently many packets exchanged between a sampled loop pair, we derive that the pair is actually vulnerable to a traffic loop attack. The percentage of sampled pairs that succeed in this final verification step helps approximate how many of the *potential* loops are *effective* loops that attackers can abuse.

4 Evaluation

In this section, we follow the five-step approach described in Section 3 to investigate loops in three protocols: DNS, NTP, and TFTP. The list of servers running DNS and NTP is provided by Shadowserver's daily report. Since Shadowserver only reports on IPv4 addresses, in this paper, we only find loop pairs from IPv4 servers. All of the experiments were performed from Jan 5 to Jan 10, 2024. In our evaluation, we observed ≈140k DNS servers, 86k NTP servers, and 22k TFTP servers that could potentially form loops.

Additionally, we investigated loops in six legacy protocols: Active Users, Daytime, Time, QOTD, Echo, and Chargen, observing 124k servers that could potentially form loops. Finally, we verify whether the IPs we found are actually vulnerable to the loop attack by using a UDP proxy to simulate the loop DoS attack. In the following, we present the loop identification and verification for each protocol in more detail.

4.1 Discovery Probes

The aim of the discovery probes is to trigger diverse responses from real servers. Ethically, we would like to design less offensive probes and minimize the number of probes. For each protocol, the discovery probes include both well-formed payloads and mutated payloads. In this section, we describe discovery probes designed for each protocol.

DNS DNS is used by various applications to, e.g., translate domain names to IP addresses. In short, controlled by a series of flags, DNS messages can be split into three major categories: query, response, and error. The DNS discovery probes cover different types of DNS messages. In total, we prepared 25 DNS discovery probes.

First, we constructed nine query-based payloads. When building query-based payloads, we first construct well-formed DNS queries using different rd (recursion desired) flags. When the rd flag is set, the server will attempt to resolve the query recursively, which may thus trigger different responses.

Next, we constructed semantically invalid queries by mutating well-formed DNS queries. For example, we constructed a DNS query payload where the question count field (qdcount) is 1, but the payload does not have an actual question field. We also constructed DNS query packets with mutated rcode fields. Normally, a DNS query should use rcode 0, which suggests no error. We instead replace the rcode with other known (e.g., 2, which indicates server failure error) or random values. We also mutate the opcode field of DNS query packets to other unused values while the packet still preserves a legit DNS query. In addition to building DNS packets with mutated header flags, we also built packets with mutated question fields. For example, we build DNS query packets asking for a DNS record of unknown qtype 280 and unknown qclass 280 instead of querying for a common 'A' record. Such mutated DNS queries can trigger server failures or format errors and thus enrich diversity.

Next, we constructed 11 response-based payloads. Similar to constructed DNS queries, the 11 DNS responses also include well-formed and semantics-violating response payloads. For example, in one of the responses, we set the question flag (qr) to zero, which indicates the payload is expected to be a DNS query, while the payload does include a well-formed answer record. Except for mutating header fields such as opcode, rcode, and ancount etc., we also built semantics-violating response payloads by adding random bytes to the payload, which can produce packets that cannot be successfully parsed.

Finally, we also include five well-formed error messages with different error type fields (rcode), including format error, server failure, not implemented, name error, and refused.

Note that the DNS discovery probes only cover basic DNS features. In particular, we did not prepare probes using extended DNS, e.g., DNS cookie and DNSSEC.

NTP NTP is a UDP-based protocol that provides clock synchronization between hosts in a network. NTP messages have eight different modes, controlled by the mode field, including reserved (mode 0), symmetric (mode 1, 2), client and server (mode 3, 4), broadcast (mode 5), NTP control message (mode 6), and private reserved (mode 7) [35]. NTP has a dedicated poll field that limits the interval between successive messages. It is thus less likely that standard NTP messages cause loops by successful requests. Consequently, when preparing the discovery probes for NTP, we mainly focused on unexpected NTP payloads that may trigger error messages from servers.

We constructed 30 such discovery probes, focusing on four different NTP modes: server (5x), broadcast/multicast (1x), NTP control messages (20x), and private reserved (4x). For server mode, we built several well-formed NTP packets with different stratum numbers (14-16) and KISS-O'-DEATH packets[1] with unregistered kiss codes. In principle, a server shall not respond to these constructed requests; they should only trigger responses from faulty implementations. In addition, we constructed one well-formed broadcast packet. Correctly implemented servers shall not respond to a broadcast message from clients; thus, the discovery probe will only trigger faulty servers. With respect to NTP control messages, we first built seven NTP error messages with defined error_code values (1 to 7) and also an NTP error message with an undefined error_code. Next, we constructed well-formed and mutated non-error NTP control messages, such as *write clock variables* and *trap response* messages. Finally, we constructed a few mutated NTP packets of private reserved mode by combining a well-formed NTP header with a dummy payload.

These four modes do not cover all NTP modes. In particular, we ignored client mode requests as they will mostly trigger

[1] KISS-O'-DEATH packets are used by NTP servers to rate-limit the NTP client's requests, which would only affect our scanner at most.

successful responses. We also did not construct any discovery probes of symmetric active or passive mode. In the symmetric mode, peers in both active and passive modes can synchronize the clock according to the received messages, which may interfere with the normal operations of servers.

TFTP TFTP is a UDP-based file transfer protocol that allows a client to read and write files on a remote host. TFTP offers five message types: read and write request, ACK, error, and data. According to RFC-1350 [42], upon receiving a request, a TFTP server chooses a transfer identifier (TID) from 0 to 65,535. In later communications, the server uses the TID as its source port. However, despite the RFC specification of using random source ports, some servers still use source port 69 to send a response. Because a TFTP server using random ports is less likely to be involved in a loop, we instead focus on those TFTP servers that use the fixed 69 port to send error messages. We designed 23 discovery probes covering all these message types except for write request.

We prepared nine query message probes. We first included well-formed read requests. Then, we also constructed read requests with mutated filename and mode fields, e.g., with a non-ASCII string mode field, which is non-compliant with the RFC. For data messages, we prepared three probes. In accordance with RFC-1350 [42], we built two well-formed data messages of sizes 512 and 0 bytes, indicating the intermediate transferring block and the last block, respectively. For the third message, we constructed a data message larger than 512 bytes, which is not allowed by the RFC. For ACK messages, we prepared two probes: one with a proper index field according to RFC-1350 [42] and one without. Finally, we constructed nine error probes: eight with standardized error codes and one with an unknown error code.

Note that TFTP requests may probe for specific files. If our requests happen to match an existing file, the server may start transferring the actual file content to us. To avoid such accidental leaks, we used a dummy filename `fJFJmcl.jieopg` in all query-based discovery probes. For a similar reason, we refrained from constructing discovery probes of write query type, as we may unintentionally create files on real servers.

Legacy Protocols We also include six legacy protocols in our analysis: QOTD (RFC 865 [6]), Chargen (RFC 864 [3]), Echo (RFC 862 [5]), Time (RFC 868) [7], Daytime (RFC 867) [4], and Active Users (RFC 866) [2]. Except for Time, which requires the client to send an empty UDP datagram, all other protocols define that servers should reply to arbitrarily-formatted requests. However, there is still a chance that Time servers do not follow RFC strictly and accept a non-empty UDP datagram. When receiving a UDP datagram, Chargen, QOTD, and Echo servers will send random data, a quote, or mirror the request, respectively. Time and Daytime servers respond with the current time (and day), and Active Users servers send a list of active system users. Thus, in principle, any two servers running these legacy protocols can form a loop pair. An attacker can initiate a loop by sending an arbitrary IP-spoofed UDP datagram to one of the servers. Thus, for all six legacy protocols, we use the same discovery probe: a UDP payload containing the single character ’a’. This way, we learn the accessible servers running each protocol.

Sending Discovery Probes After defining the discovery probes, we send them to servers of the respective protocols. We got the list of seed server IPs running the tested protocols from Shadowserver’s daily report. The daily report includes 2,009,205 IPs for DNS and 1,712,434 IPs for NTP. As discussed previously, for TFTP, we only focused on TFTP servers that send error messages from port 69. Thus, instead of using Shadowserver’s daily report, we performed a full IPv4 scan and found 148,676 IPs for TFTP. For the six legacy protocols, we performed a full IPv4 scan and found 3,238 Active Users servers, 13,983 Daytime servers, 13,195 Time servers, 52,707 Echo servers, 22,576 Chargen servers, and 17,927 QOTD servers. Since we use a non-empty UDP datagram as the discovery probe for Time servers, the discovered servers would accept a non-empty UDP datagram and are thus potentially vulnerable to the loop attack. For each protocol, we then use ZMap [48] to send the discovery probes to all seed servers of the given protocol. For each host, we wait for (at least) 10 seconds before proceeding with the next probe to avoid high query loads to real servers. We defer to Section 6 for a detailed discussion of our scanning setup.

We received at least one reply from 1,366,247 (68%) DNS servers, 1,626,754 (95%) NTP servers, and all 148,676 TFTP servers. For DNS, the response rates are significantly lower than for NTP. We speculate this is due to more aggressive IP address churn among DNS servers. Previous research also reported IP churn behavior for DNS servers, where the number of accessible IP addresses dropped to 60% percent after one day [29]. Similarly, Kührer et al. noted a significantly lower churn rate for NTP compared to DNS (Table 3 in [31]). To counter the IP address churn, we could perform full Internet scans ourselves — while causing higher loads to the Internet.

4.2 Response Clustering

We now describe how we cluster responses for each protocol.

DNS We collected 327,841 unique DNS responses via the discovery probes. We parsed each unique DNS payload according to the DNS packet structure. In the packet, we considered the opcode field and the 8-bit flag field (QR, AA, TC, RD, RA, Z) as important fields, and a payload is first categorized using these fields. Second, if the payload was a DNS response packet (QR flag is 1), we further categorized the payload according to its rcode (response code) field. Third, we categorized payloads according to the number of records they contained, indicated by qdcount, ancount, nscount, and arcount fields. These field values can range from 0 to 65,535, so we created five categorization buckets of 0, 1, 2-256, 257-8,192, and 8,193-65,535. To our surprise, when inspecting collected responses, we found that some DNS servers sent a DNS response packet where the question field domain name did not match the domain name used in the discovery probes. Thus, we also categorized payloads based on whether the domain name in the QD or AN field matched the domain name we used in the discovery probes. We ignored the transaction ID and IPs during clustering, as they cannot change the semantics of a payload.

Among the collected responses, we also identified responses that cannot be parsed normally. However, such responses may still contain a complete or partial header. For these responses, we first categorized them as abnormal responses and then attempt to parse the DNS header byte-by-byte to the extent possible. The semi-parsed responses were then categorized using the extracted header information.

After clustering, we identified 90 clusters with more than 10k unique servers sending corresponding responses.

NTP We collected 8,459,730 unique NTP responses, all of which we attempted to parse. In the packet, we considered LI (leap indicator), VN (version number), stratum, poll, and mode as important fields. A payload would first be clustered according to these fields. Then, packets that were NTP control messages (mode 6) underwent further categorization. Unlike normal NTP headers, control message headers contain R, E, M flags, opcodes, and status fields — all of which we considered important categorization features. We ignored identifier, authenticator, and timestamps fields, as they should not affect the semantics of a packet.

Note that our NTP clustering approach only focused on NTP control messages. Other types of NTP messages can have more distinct fields. For example, NTP private reserved mode messages contain additional error and request code fields. By parsing NTP messages as common NTP headers and extracting stratum, leap fields, etc., we may omit or misinterpret some of such distinct fields.

Overall, we identified 55 NTP clusters with more than 10k unique servers sending corresponding responses.

TFTP We collected 533 unique responses from TFTP servers. The majority of responses were different types of error messages. Thus, our clustering for TFTP payloads mainly focused on the type of error. We first clustered the TFTP payloads according to their opcode, and if the TFTP payload is of error message type (opcode=0005), we further clustered them according to the errorcode field. As we inspected malformed error message responses from real servers, we categorized an error message according to syntactic correctness. For example, we clustered error messages by observing whether the error message correctly ends with a single NULL byte.

With this approach, we found six clusters with more than 10k unique servers sending corresponding responses.

Legacy Protocols Given that legacy protocols reply to arbitrary request payloads, we simply group all responses in a single catch-all cluster.

4.3 Loop Probe

As described in Section 3.6, for each non-legacy protocol, we sample five payloads per cluster and use ZMap to send them to all servers. For DNS, NTP, and TFTP, the number of sampled probes is 299, 248, and 18, respectively. This is less than five times the number of clusters because some clusters have fewer than five payloads. To further reduce the load of these additional probes, we increased the cooldown time between different probes per host to 15 seconds.

We then clustered the probed responses. To confirm the effectiveness of our clustering approach, as introduced in Section 3.6, we verified the number of IPs responding to different numbers of sampled probes in the same cluster.

For TFTP, the clustering performed well, as all six clusters had the majority of servers respond to all sampled payloads.

Regarding NTP, out of all 55 clusters, 44 clusters performed well. We treated the 11 problematic clusters as follows. Seven of them only received responses from less than a thousand servers; we dropped these clusters directly. For the other four, we also did not optimize the clustering as we found that these clusters are not involved in any potential cycles anyway.

For DNS, out of all 90 clusters, 88 clusters performed well, i.e., the majority of servers replied to all five sampled probes. The two bad clusters are DNS *response* packets containing a domain name that does not match the domain name we used in the discovery probe. In the two bad clusters, probe payloads containing `graph.facebook.com` trigger significantly more servers' responses than other sampled payloads. To understand why, we investigated the collected responses when using payloads containing `graph.facebook.com`. We believe the responses were generated by censorship DNS middleboxes, as in the answer field, the domain is resolved to a local IP `10.10.34.35`. A previous report recognized such responses as an attempt to block users' access by Iran [20]. As discussed in Section 3.6, we updated the clustering approach introduced in the previous section. We removed sampled payloads containing domain `graph.facebook.com` from the two bad clusters and manually added them to two new clusters.

We do not consider loop probes for legacy protocols as we expect identical behavior irrespective of the probes we send.

4.4 Loop Graph and Loop Search

So far, we gathered all the essential information to find potential loop pairs among real servers. Figure 4 shows the resulting loop graphs for DNS, NTP, and TFTP. Recall that

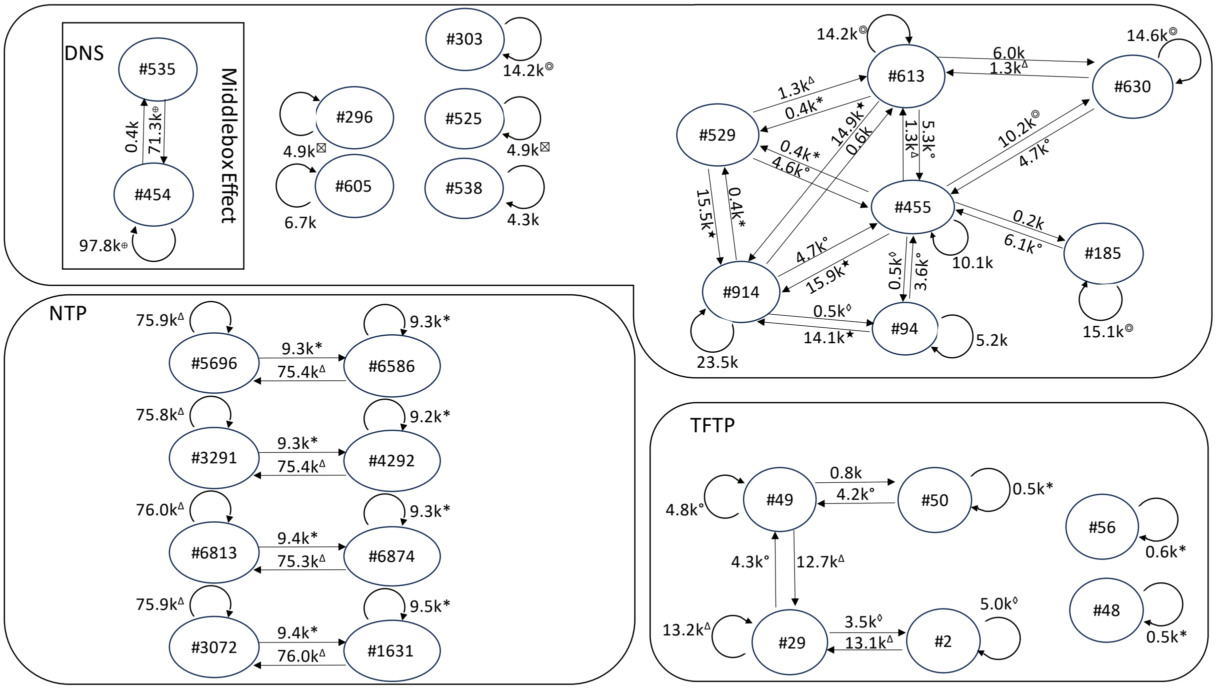


Figure 4: Loop Graph. Nodes in the graph represent payload clusters; Appendix B provides example payloads for several clusters. The edge from a node #A to another node #B indicates hosts that would send a #B response when receiving a #A payload, where the edge label is the number of servers with such behavior. For each protocol, edges labeled with the same symbols (*, ○, ◎, △, ☒, ★, ◊, ⊕) indicate that the majority of the server IPs (> 70%) corresponding to these edges overlap.

nodes represent cluster IDs and edges from cluster c_i to cluster c_m are hosts with response function $F(c_i) = c_m$. To find all possible loop pairs per protocol, we searched for cycles up to length $n = 4$. When inspecting loops, we filtered out loops having edges corresponding to less than 100 IP addresses. In the following, we report on the loop pairs per protocol.

DNS For DNS, we identified 140,334 (10.2%) servers that can be involved in a loop. In this section, we focus on 24 identified major cycles. In DNS, the two largest cycles we identified, cycle {454, 535, 454} and cycle {454, 454}, affect 98,669 IPs. Recall from the previous section that the two largest cycles are caused by the newly identified middlebox-effect clusters (clusters 454 and 535).

In the two middlebox-related cycles, we found 89,040 (90.2%) affected IPs are from Iran, and 9,629 (9.8%) IPs are from other countries, e.g., China, Indonesia, Russia, etc. In particular, the number of affected Iranian IPs covers 93.1% of all known Iranian open resolvers from Shadowserver's daily report. Since clusters 454 and 535 are both DNS responses that DNS servers shall not react to, the high IP affect rate is a clear indicator that it is a nationwide reaction by Iran's middleboxes to inject DNS responses when observing clusters 454 and 535 packets. However, we did not observe the same behavior for affected IPs in other countries, though some countries, e.g., China [23], are known for having DNS censorship. In other words, affected IPs in other countries are not necessarily due to country-level middleboxes. Instead, vulnerable IPs outside of Iran are likely actual vulnerable DNS resolvers. In the end, even if some of the identified IPs are actual vulnerable DNS resolvers, they can still be paired with middlebox-effect IPs in Iran to form loop pairs.

For cycle {454, 535, 454}, upon receiving a response packet for a censored domain (e.g., graph.facebook.com) with many NS records (cluster 535), middleboxes or servers on edge [535, 454] inject a response packet containing only the answer for the censored domain (cluster 454). When other middleboxes or vulnerable servers (on edge [454, 535]) observe a response packet containing a censored domain, they instead inject a response containing multiple NS records. For cycle {454, 454}, it appears that a short response for a censored domain would again trigger middleboxes or vulnerable servers to inject responses for the censored domain. Previous research shed light on weaponizing HTTP middleboxes in DDoS attacks [14]. We show that the DNS middleboxes also have similar behavior that attackers can abuse for loop DoS attacks.

Note that our measurement for DNS middlebox-related loops has two limitations:

- Our scanner's location can impact the number of identified loop pairs. In the two middlebox-related loops, the majority of affected IPs are from Iran. These loops are caused by Iran middleboxes' reactions to DNS response packets with a censored domain. According to previous research [12, 47], countries with DNS censorship tend to deploy middleboxes in border ASes. Since our scanner was located outside of Iran, when scanning open resolver IPs in Iran, the probe (e.g., of type 454) would go through border ASes and trigger middleboxes. However, when pairing two IPs in Iran, as the DNS packet is never routed to border ASes, the DNS middlebox would not inject forged responses to cause loops. As a result, we might overestimate the number of loop pairs for middlebox-related loops whenever we pair two IPs in Iran. Having said this, we can still pair vulnerable IPs in Iran with vulnerable IPs in other countries, which would cross the border ASes. When excluding Iranian-only pairs, we can still form 811.15M loop pairs.

- Given that we used a list of open resolvers provided by Shadowserver to perform the scan, we may also underestimate the number of affected IPs in middlebox-related loops. In principle, when pairing with vulnerable hosts in other countries, any Iranian IP can be used as a loop host, as the censorship middlebox on the path will be involved to inject responses. To show the potential underestimation, we perform a preliminary experiment. We picked three IPv4 blocks in Iran; each contains 65,536 IPs. Next, we send type 454 probes to all IPs in the three blocks. In total, we received type 454 responses from 193,239 (98.3%) IPs. However, Shadowserver's daily report only contains 10,946 (5.6%) open resolver IPs in the three IPv4 blocks.

Excluding the two middlebox-related loops, all other cycles affect 56,239 IPs in total. Among non-middlebox-related cycles, we observed server failure error message loops. When servers involved in these cycles received a DNS server failure response, they would reply with another DNS server failure response to form a loop. Cycles {630, 630}, {303, 303}, and {613, 613} are all loops caused by server failure error packets, affecting 14.6k, 14.2k, and 14.2k servers, respectively. Server failure responses in the three clusters (630, 303, 613) differ by whether the server supports recursion or not (flag ra is 0 or 1) and whether the server failure response includes an answer record. As shown in Figure 4, the majority of servers in these three cycles therefore overlap, which is a strong indicator that the problem is caused by shared faulty implementation that misinterprets a server failure message.

Interestingly, we also observed many loops triggered by DNS responses in cluster 455, i.e., common DNS responses to the domain controlled by us. Such loops can be classified into two categories. In the first category, a response to the domain controlled by us triggers an error message, and then the error message again results in a response to our controlled domain. Cycles {455, 630, 455} and {455, 613, 455} are this type of cycle, and they affect 15.0k and 6.6k IPs, respectively.

In the second category, a response to the domain controlled by us triggers a well-formed response with slight differences in flag fields, and the mutated response again triggers a response to our controlled domain. Cycles {455, 94, 455}, {455, 185, 455}, {455, 914, 455}, and {455, 529, 455} are all loops of this category. Compared to responses from cluster 455, payloads from clusters 94, 185, and 914 only differ from cluster 455 by *rd* (recursion desired), *ra* (recursion available), and *aa* (authoritative answer) flags, while payloads from cluster 529 contain additional resource records and NS records.

For all cycles in the two categories, each has an edge [X, 455], where X $\in$ {630, 613, 94, 185, 914, 529}. The majority of servers on edge [X,455] overlap, which indicates a faulty implementation used by servers that would accept and resolve an incoming response packet.

In addition to the clusters introduced previously, there are also several sets of IPs performing error message loops (e.g., {613, 630, 613}), legit response loops (e.g., {914, 94, 914}), and error message to legit response loops (e.g.,{613, 914, 613}).

Finally, we also identified several self loop cycles {296, 296}, {538, 538}, {605, 605}, and {525, 525}. Clusters 296 and 605 are both format error messages, where they differ by the *ra* (recursion available) flag. Clusters 538 and 525 are legit DNS responses to our domain, where they differ by the number of NS records in the DNS response.

NTP For NTP, we identified 12 cycles with 86,469 (5.3%) hosts potentially affected. As discussed in Section 4.1, for NTP, we expected to identify error message loops. As per our expectation, all NTP cycles are error message loops. Shown in Figure 4, NTP loops correspond to eight different clusters. The eight clusters are all NTP payloads of mode 7 (private reserved mode). Payloads of the eight clusters differ by the NTP version (v2 or v3) field and the implementation field (the poll field when parsed as an NTP header).

Shown in Figure 4, the same set of servers transform an NTPv3 error message (clusters 5696, 3291, 6813, 3072) to an NTPv2 error message (clusters 4292, 6874, 6586, 1631). Similarly, there is also a set of servers that transform an NTPv2 error message to an NTPv3 message. This suggests that there are sets of NTPv3 and NTPv2 servers running faulty software that would reply to an error message. Indeed, CVE-2009-3563 [38] describes an error message loop in ntpd [1], a popular NTP server implementation. The CVE shows that before ntpd v4.2.6, an attacker can cause an error message loop between ntpd servers using malformed mode 7 NTP packets. Thus, we believe that some of the loops we identified could be explained by outdated ntpd servers. The same set of faulty NTPv3 servers and NTPv2 servers can also form a loop when receiving an error message from a server running the same version, e.g., cycle {5696, 5696}.

TFTP For TFTP, we identified 9 cycles with 22,518 (15.1%) potentially affected hosts in total. These 9 cycles hint at faulty servers that react to certain error messages and respond with another error message. Although the TFTP standard explicitly states that an error message "is not acknowledged," we still observed many TFTP servers diverging from this. In fact, *all* 9 TFTP cycles are error message loops.

As shown in plot Figure 4, TFTP loops are constructed from six clusters. Clusters 56, 49, and 29 are error messages of unknown transfer ID (error code 5), illegal TFTP operation (error code 4), and not defined (error code 0) types. Cluster 50 is slightly different from cluster 49. It is also error messages of illegal TFTP operation type; however, the messages feature an additional NULL byte at the end. Similarly, cluster 48 is also an illegal TFTP operation error message, but it does not end with a NULL byte. Different from other clusters, cluster 2 is a TFTP message with opcode 0. Opcode 0 is not specified in RFC [42]. However, as some TFTP payloads in cluster 2 contain an "invalid request" message, we believe these payloads are likewise error messages.

Legacy Protocols We omit the loop graphs and cycle search for the legacy protocols. We assume that each server of any legacy protocol can create a cycle with an arbitrary other server of the same (or even another) protocol.

4.5 Geo-Distribution

Before probabilistically verifying the potential loop pairs, we now provide a first analysis of ASes hosting potential looping servers. For all six non-legacy protocols, DNS, NTP, and TFTP, we found thousands of affected ASes overall. Figure 5 summarizes the geographical and topological breakdown of the affected servers for each non-legacy protocol.

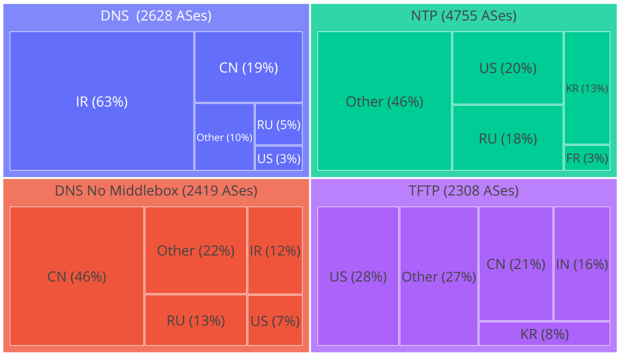


Figure 5: AS and geo-distribution of vulnerable servers.

For DNS, Iran and China have the most vulnerable DNS servers. The majority of affected IPs in Iran are involved in middlebox-related cycles discussed previously. In Figure 5, we also provide the analysis result when excluding the two middlebox-related cycles. When including or excluding middlebox-related cycles, AS `TCI` in Iran and AS `Chinanet-backbone` host the most vulnerable DNS servers, 72,026 and 7,191 IPs, respectively.

Regarding NTP, the US, Russia, and Korea host most vulnerable NTP servers. In particular, AS `Korea Telecom` alone hosts 5,018 affected NTP servers.

For TFTP, the US, China, and India are the top three sources of affected servers. Out of all ASes, the AS `Frontier-FRTR` hosts the most faulty TFTP servers (2,559).

4.6 End-to-End Loop Verification

In the loop graph, we identified *potential* cycles according to the response functions of each server. Next, we *verified* these loops in controlled end-to-end loop experiments following the approach proposed in Section 3.8.

In the experiment, we used the proxy to verify 100 sampled loop pairs from each cycle we identified. As one IP can be involved in several cycles, the same IP can show up in different sampled pairs. To avoid flooding real servers, we applied a rate limit per IP instead of per loop pair. The rate limit used by the proxy is 3 pps. To avoid sending too many packets to the sampled servers, the proxy would stop delegating packets once it found more than 25 packets sent between a loop pair. Finally, we checked the number of sampled loop pairs in each cycle that successfully reached a 25-packet loop.

Cycle	Loop IPs	Loop Pairs	Success Rate
{1631, 1631}	9.5 k	45.02 M	94%
{4292, 4292}	9.2 k	42.46 M	94%
{6874, 6874}	9.3 k	42.82 M	96%
{6586, 6586}	9.3 k	42.79 M	93%
{3291, 3291}	75.8 k	2872.10 M	93%
{6813, 6813}	76.0 k	2886.59 M	93%
{5696, 5696}	75.9 k	2878.17 M	98%
{3072, 3072}	75.9 k	2884.01 M	97%
{3291, 4292, 3291}	84.7 k	703.83 M	99%
{5696, 6586, 5696}	84.7 k	704.59 M	94%
{6874, 6813, 6874}	84.7 k	709.85 M	94%
{3072, 1631, 3072}	85.4 k	714.40 M	93%
Overall (estim.)	82.3 k	3.47 B	(avg) 95.2%

Table 1: NTP verification summary. For each cycle, we reported the number of potentially affected IPs, pairs, and the verification success rate. The last row (in bold) provides an estimation for the total number of *confirmed* IPs and pairs based on the weighted average of verification success rates.

In Table 1, Table 2, and Table 3, we provided the detailed verification results for each cycle we identified for NTP, DNS, and TFTP, respectively. For these three protocols, we performed the proxy verification within approximately three hours after the scanning. Overall, we found about 200k servers that can be abused in loops, giving attackers the opportunity to create billions of loop pairs.

For TFTP and NTP, the average verification rates were 84.5% and 95.2%, respectively. Combining the number of affected IPs from the loop graphs, we estimate that 82,318 (5.1%) IPs and 3.47B pairs for NTP and 19,027 (12.8%) IPs and 0.18B pairs for TFTP can be involved in a loop attack.

For DNS, we verified middlebox-related cycles[2] and non-middlebox-related cycles separately, and the verification success rates were 75.6% and 65.5% respectively. We observed that the verification success rates for some cycles, e.g., {538, 538}, {605, 605}, and {296, 296} are lower: 16%, 25%, and 24% respectively. In order to reason about this discrepancy, we observed the number of packets sent between each sampled loop pair. If we only observed zero or one packet among a loop pair, it mostly suggested at least one server in the loop pair was no longer accessible. Instead, if we observed more than two packets (while less than 25 packets) between a loop pair, the loop was mostly stopped because of the rate limit of at least one server in the loop pair. Out of the 100 sampled loop pairs, we found 83 failed pairs for cycle {538, 538}, 52 failed pairs for {605, 605}, and 42 failed pairs for {296, 296} with less than two packets sent. Thus, the low verification success rate for the three cycles was likely due to frequent IP address churn. Note that our proxy server only passively delegates packets while not verifying if the responses generated by servers are of the expected types, thus non-deterministic server behaviors can impact our verification. Indeed, a DNS server may behave differently against the same payload during the loop probe and the verification, e.g., due to caching. One can tackle such non-determinism by precaching the requested data or by performing the loop probe multiple times to filter out servers with non-deterministic behaviors. Overall, the verification success rate for DNS, including both non-middlebox-related cycles and middlebox-related cycles, was 70.1%. According to the number of affected IPs reported by the loop graph, we believe that 98,374 IPs (7.2%) and 1.01B pairs for DNS can be involved in a loop attack.

For legacy protocols, any of the discovered servers can be paired with others to form a loop pair. We thus sampled and verified 1000 random host pairs for each legacy protocol. For legacy protocols, the proxy verification is performed approximately 12 hours after the scanning. In Table 4 (see Appendix A), we summarized the verification success rate for the six legacy protocols: Active Users (67.5%), Daytime (79.6%), Time (78.8%), Echo (83.8%), Chargen (40.3%), and QOTD (87.8%). Of all legacy protocols, Chargen had the lowest verification success rate. Similar to DNS, we found that majority of failed pairs were caused by unreachable servers.

5 Discussion

We now discuss important aspects with respect to how attackers can accelerate loops or find cross-protocol loops, and describe mitigation strategies for the identified vulnerabilities.

5.1 Self-Amplifying Loops

In our experiment, we found some servers, e.g., Chargen (1.9k), sending more than one response upon receiving one probe. The majority of these servers sent two to nine responses upon receiving a probe, while there were some servers that sent dozens or even hundreds of responses. Such multi-response behaviors can be caused by network issues or faulty software. Either way, if such servers can be involved in a loop pair, the number of parallel loops could grow exponentially. Interestingly, 73.3% of these servers were concentrated in just three ASes, which further increases the threat of a loop-based DoS attack for the respective ASes.

5.2 Cross-Protocol Loops

The main focus of this work is on identifying loops among servers running the *same* UDP-based application-layer protocol. Unfortunately, though, attackers can also trigger loops *across* protocols. For example, DNS, NTP, TFTP, and other protocols implement format error messages. When a server receives a malformed request, it responds with a format error message to notify the client or help debugging. However, for two servers running different UDP protocols, they lack knowledge of the packet format of each other. Thus, such format error messages may enable cross-protocol loops.

We showcase such a cross-protocol loop as a concrete example with TFTP and DNS. Because unexpected packets from other protocols may crash server programs, we refrained from using the same methodology to discover cross-protocol loops among real servers. Instead, we emulated the cross-protocol loop in a local environment. We set up a local DNS server using bind9 [13] and a local TFTP server using erlang-tftp [19]. To set up the TFTP server, we used the example instruction in [19] to start a default TFTP server. For the DNS server, we used the default configuration of bind9. By sending a *TFTP not defined* error message to the DNS server, we successfully caused a loop between the two servers. bind9 interprets the TFTP message as a malformed DNS packet and replies with a *format error* message to the TFTP server. Upon receiving the format error response, the TFTP server finds the packet has an unknown error code and considers it as a malformed TFTP packet. Thus, the TFTP server sends a *not defined* error response to the DNS server and forms a loop. In Section 4.4, we showed that DNS and TFTP servers sending such messages are not uncommon, underlining that cross-protocol are not only a theoretical threat.

[2] As discussed in Section 4.4, when verifying middlebox-related cycles, we only pair IPs from Iran with IPs from other countries.

The six legacy protocols explored in the paper can also participate in cross-protocol loops. For example, a TFTP server and a Chargen server can form a loop pair easily. The Chargen server's dummy response would trigger the TFTP server's not defined error, and the error message would again trigger the Chargen server's response to sustain the loop. We confirmed such a loop is possible by using a local setup of erlang-tftp and xinetd's Chargen service [45], and we can form a loop by sending a dummy UDP payload to the Chargen server.

5.3 Automating Discovery Probe Generation

In this paper, we provide a methodology of identifying application-layer traffic loops. The discovery probe design requires manual efforts, which can limit the number of identified loops. In this section, we briefly discuss potential ways of automating the discovery probe generation. To automatically generate mutated payloads, fuzzing is a viable approach. Fuzzing would thus provide an even more systematic way to fully explore the threat landscape. However, fuzzing-generated discovery probes would require additional checks or strict restrictions to avoid crashing real servers. For example, when the length is not restricted, fuzzing can generate a long mutated payload, which is more likely to cause memory corruption. Next, fuzzing would also likely generate many semantically identical discovery probes. To reduce the number of probes, one could use our proposed clustering methodology to sample discovery probes from each cluster.

5.4 Loop Mitigation

We now discuss mitigation steps that developers and server operators can take to reduce the risk of loops.

Do not respond to errors Most loops are caused by servers reacting to error messages or responses from other servers. According to RFCs for all studied non-legacy protocols, a server should not reply to any error messages or responses generated by other servers. Enforcing this recommendation would mitigate the vast majority of loops. Recall that many loops we identified share the same set of faulty servers. Hopefully, we can eliminate a large number of loop pairs and vulnerable hosts by patching these vulnerable servers.

Suppress errors While error messages are useful for debugging and sometimes inherently important for protocols to function correctly, they do increase chances of loops. Recall the TFTP + DNS cross-protocol loop. When receiving a DNS error message packet, a TFTP server generates an error message in TFTP format, which is considered a malformed DNS packet when received by a DNS server. As servers do not know other protocols, cross-protocol loop attacks are rather hard to mitigate. One effective mitigation is to make servers stay silent when receiving "malformed" packets.

Rate limiting Servers can mitigate loops by applying rate limits. When the number of packets among two victim servers reaches the rate limit, the packet loop terminates. However, rate limits cannot mitigate the loop attack completely. For example, DNS resolutions can take a considerably long time, which can help an attacker to evade rate limits. Furthermore, rate limits may only degrade the packet loop but do not entirely stop it. In fact, the attacker can periodically send the loop-triggering payloads to victim servers to restart them.

Source port validation Loops always span two servers and their corresponding UDP server ports (e.g., 53 for DNS). Loops could be stopped if servers expect all benign requests to come from *different* ports, e.g., out of the ephemeral client UDP port range. This is an elegant and easy fix that likely works for most clients. The same logic can be applied for network-level filtering of attack traffic. That is, normally (at least) one of the two UDP ports in a given UDP stream come from the client-chosen ephemeral range. But note that this solution might break the compatibility of some software. For example, a popular NTP server software, ntpd [1], would, by default, use source port 123 to send symmetric mode messages to other NTP servers. Legacy DNS servers may also send requests from source port 53.

Quality of service To mitigate loops, network operators can give less preference to abused protocols to drop attack packets in case of network congestion. In particular, the UDP ports of the legacy protocols can be assigned low QoS priority. Non-legacy protocols (TFTP, DNS, NTP) instead require case-by-case decisions to minimize interference with normal services.

6 Disclosure and Ethics

We now discuss our disclosure process and revisit the ethical aspects of our methodology, experimental setup, and findings.

Disclosure We started the disclosure process of our work's findings in December 2023. Though we found loops in several protocols, there is no easy way to learn the vulnerable software and products that cause the loop behavior.

Fortunately, Shadowserver provided us with the fingerprints of vulnerable hosts. In parallel, we also disclosed the list of affected IPs to ASes. Some of the affected ASes provided us with information on affected products and software. The vulnerability is mostly caused by proprietary software used by vendors. As of Feb 2024, several vendors confirmed loop vulnerabilities: Broadcom (DNS), Honeywell (NTP), Mikrotik (TFTP), Brother (TFTP), and Microsoft (TFTP).

There are also end-of-life products from other vendors that are likely affected. For example, Cisco 2800/2970 routers can likely form an NTP loop, and Zyxel's old firewall products, e.g., ZyWall 2, is seemingly affected by a DNS loop.

Ethics We carefully designed our experiments according to ethical principles. For each protocol, we used hand-vetted discovery probes to minimize the probability of interfering with real servers. During the scanning, we followed general best practices [48] and also minimized the number of probes to servers via clustering and applied rate limits to avoid causing noticeable network or computational loads. Although we offered an opt-out option, we did not receive a single such request throughout our experiments, which underlines the low-rate and defensive nature of our scans.

7 Conclusion

We introduced a systematic approach for finding and verifying application-layer packet loops among real servers. Such loops can be abused in various attacks, e.g., link flooding and server resource exhaustion, and only require a limited attacker ability, i.e., to send IP-spoofed packets to initiate loops. These loops are prevalent among network infrastructures, with our work revealing billions of pairs that can be abused by attackers. Empirically, we also find packet loops among servers across protocols, which can affect even more real servers in practice. We hope this work can raise developers' and protocol designers' awareness of packet loops and help in patching vulnerable software and updating protocol standards.

Availability

Our project is available on https://github.com/cispa/loop-DoS.

References

[1] Ntpd - Network Time Protocol (NTP) Daemon. https://docs.ntpsec.org/latest/ntpd.html, accessed on October 6, 2023.

[2] Active Users Protocol. RFC 866, May 1983.

[3] Character Generator Protocol. RFC 864, May 1983.

[4] Daytime Protocol. RFC 867, May 1983.

[5] Echo Protocol. RFC 862, May 1983.

[6] Quote of the Day Protocol. RFC 865, May 1983.

[7] Time Protocol. RFC 868, May 1983.

[8] Florian Adamsky, Syed Ali Khayam, Rudolf Jäger, and Muttukrishnan Rajarajan. P2P File-Sharing in Hell: Exploiting BitTorrent Vulnerabilities to Launch Distributed Reflective DoS Attacks. In *USENIX Workshop on Offensive Technologies*, Washington, DC, USA, 2015. https://www.usenix.org/conference/woot15/workshop-program/presentation/p2p-file-sharing-hell-exploiting-bittorrent.

[9] Aflnet. AFLNet: A Greybox Fuzzer for Network Protocols. https://github.com/aflnet/aflnet, accessed on October 6, 2023.

[10] Marios Anagnostopoulos, Stavros Lagos, and Georgios Kambourakis. Large-scale Empirical Evaluation of DNS and SSDP Amplification Attacks. *Journal of Information Security and Applications*, 66:103168, 2022.

[11] Anonymous. Operation Global blackout. https://pastebin.com/NKbnh8q8, accessed on October 6, 2023.

[12] Anonymous. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *USENIX Workshop on Free and Open Communications on the Internet*, San Diego, CA, 2014. https://www.usenix.org/conference/foci14/workshop-program/presentation/anonymous.

[13] Bind9. Bind9 - Debian Wiki. https://wiki.debian.org/Bind9, accessed on October 6, 2023.

[14] Kevin Bock, Abdulrahman Alaraj, Yair Fax, Kyle Hurley, Eric Wustrow, and Dave Levin. Weaponizing Middleboxes for TCP Reflected Amplification. In *USENIX Security Symposium*, pages 3345–3361, Virtual Event, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/bock.

[15] Jonas Bushart and Christian Rossow. DNS Unchained: Amplified Application-Layer DoS Attacks Against DNS Authoritatives. In *Research in Attacks, Intrusions, and Defenses*, pages 139–160, Heraklion, Crete, Greece, 2018. https://doi.org/10.1007/978-3-030-00470-5_7.

[16] CERT. CERT Advisory CA-1996-01 UDP Port Denial-of-Service Attack. https://vuls.cert.org/confluence/display/historical/CERT+Advisory+CA-1996-01+UDP+Port+Denial-of-Service+Attack, accessed on January 9, 2024.

[17] CERT. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks. https://vuls.cert.org/confluence/display/historical/CERT+Advisory+CA-1996-21+TCP+SYN+Flooding+and+IP+Spoofing+Attacks, accessed on October 6, 2023.

[18] Jianjun Chen, Xiaofeng Zheng, Hai-Xin Duan, Jinjin Liang, Jian Jiang, Kang Li, Tao Wan, and Vern Paxson. Forwarding-Loop Attacks in Content Delivery Networks. In *Network and Distributed System Security Symposium*, San Diego, California, USA, 2016. https://www.ndss-symposium.org/wp-content/uploads/2017/09/forwarding-loop-attacks-content-delivery-networks.pdf.

[19] Erlang-tftp. Erlang-tftp Instructions. https://www.erlang.org/docs/23/apps/tftp/getting_started.html, accessed on October 6, 2023.

[20] Arturo Filastò, Maria Xynou, and Nima Fatemi2020-03-04. Iran Temporarily Blocks the Farsi Language Edition of Wikipedia. https://ooni.org/post/2020-iran-blocks-farsi-wikipedia/, accessed on October 6, 2023.

[21] J. J. Garcia-Luna-Aceves. Eliminating Routing Loops and Oscillations in BGP Using Total Ordering. In *IEEE Conference on Local Computer Networks*, pages 9–17, Edmonton, AB, Canada, 2022. https://doi.org/10.1109/LCN53696.2022.9843706.

[22] João José Costa Gondim, Robson de Oliveira Albuquerque, Anderson Clayton Alves Nascimento, Luis Javier García-Villalba, and Tai-Hoon Kim. A Methodological Approach for Assessing Amplified Reflection Distributed Denial of Service on the Internet of Things. *Sensors*, 16:1855, 2016.

[23] Nguyen Phong Hoang, Arian Akhavan Niaki, Jakub Dalek, Jeffrey Knockel, Pellaeon Lin, Bill Marczak, Masashi Crete-Nishihata, Phillipa Gill, and Michalis Polychronakis. How Great is the Great Firewall? Measuring China's DNS Censorship. In *USENIX Security Symposium*, pages 3381–3398, Virtual Event, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/hoang.

[24] Mattijs Jonker, Alistair King, Johannes Krupp, Christian Rossow, Anna Sperotto, and Alberto Dainotti. Millions of Targets under Attack: a Macroscopic Characterization of the DoS Ecosystem. In *Internet Measurement Conference*, pages 100–113, London, United Kingdom, 2017. https://doi.org/10.1145/3131365.3131383.

[25] Min Suk Kang, Virgil D. Gligor, and Vyas Sekar. SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks. In *Network and Distributed System Security Symposium*, San Diego, California, USA, 2016. https://www.ndss-symposium.org/wp-content/uploads/2017/09/spiffy-inducing-cost-detectability-tradeoffs-persistent-link-flooding-attacks.pdf.

[26] Min Suk Kang, Soo Bum Lee, and Virgil D. Gligor. The Crossfire Attack. In *IEEE Symposium on Security and Privacy*, pages 127–141, Berkeley, CA, USA, 2013. https://doi.org/10.1109/SP.2013.19.

[27] Daniel Kopp, Christoph Dietzel, and Oliver Hohlfeld. DDoS Never Dies? An IXP Perspective on DDoS Amplification Attacks. In *Passive and Active Measurement*, pages 284–301, Virtual Event, 2021. https://doi.org/10.1007/978-3-030-72582-2_17.

[28] Maciej Korczynski and Yevheniya Nosyk. Source Address Validation. *CoRR*, abs/2301.09952, 2023.

[29] Johannes Krupp, Mohammad Karami, Christian Rossow, Damon McCoy, and Michael Backes. Linking Amplification DDoS Attacks to Booter Services. In *Research in Attacks, Intrusions, and Defenses*, pages 427–449, Atlanta, GA, USA, 2017. https://doi.org/10.1007/978-3-319-66332-6_19.

[30] Jan Kucera, Ran Ben Basat, Mário Kuka, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. Detecting Routing Loops in the Data Plane. In *International Conference on emerging Networking EXperiments and Technologies*, pages 466–473, Barcelona, Spain, 2020. https://doi.org/10.1145/3386367.3431303.

[31] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium*, pages 111–125, San Diego, CA, USA, 2014. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kuhrer.

[32] Soo Bum Lee, Min Suk Kang, and Virgil D. Gligor. CoDef: Collaborative Defense against Large-Scale Link-Flooding Attacks. In *Conference on emerging Networking Experiments and Technologies*, pages 417–428, Santa Barbara, CA, USA, 2013. https://doi.org/10.1145/2535372.2535398.

[33] Robert Lemos. Largest-Ever DDoS Campaign Demonstrates Danger of New Attack Method. https://www.eweek.com/security/largest-ever-ddos-campaign-demonstrates-danger-of-new-attack-method/, accessed on October 6, 2023.

[34] Matthew J. Luckie, Robert Beverly, Ryan Koga, Ken Keys, Joshua A. Kroll, and kc claffy. Network Hygiene, Incentives, and Regulation: Deployment of Source Address Validation in the Internet. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Conference on Computer and Communications Security*, pages 465–480, London, UK, 2019. https://doi.org/10.1145/3319535.3354232.

[35] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[36] Giovane C. M. Moura, Sebastian Castro, John Heidemann, and Wes Hardaker. TsuNAME: Exploiting Misconfiguration and Vulnerability to DDoS DNS. In *ACM Internet Measurement Conference*, page 398–418, Virtual Event, 2021. https://doi.org/10.1145/3487552.3487824.

[37] Yevheniya Nosyk, Maciej Korczynski, and Andrzej Duda. Routing Loops as Mega Amplifiers for DNS-Based DDoS Attacks. In *Passive and Active Measurement*, pages 629–644, Virtual Event, 2022. https://doi.org/10.1007/978-3-030-98785-5_28.

[38] NVD. NVD - CVE-2009-3563. https://nvd.nist.gov/vuln/detail/cve-2009-3563, accessed on October 6, 2023.

[39] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nicholas Weaver, and Vern Paxson. Global Measurement of DNS Manipulation. In *USENIX Security Symposium*, pages 307–323, Vancouver, BC, Canada, 2017. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/pearce.

[40] Christian Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Network and Distributed System Security Symposium*, San Diego, California, USA, 2014. https://www.ndss-symposium.org/wp-content/uploads/2017/09/01_5.pdf.

[41] João Luis Sobrinho, David Fialho, and Paulo Mateus. Stabilizing BGP through Distributed Elimination of Recurrent Routing Loops. In *IEEE International Conference on Network Protocols*, pages 1–10, Toronto, ON, Canada, 2017. https://doi.org/10.1109/ICNP.2017.8117560.

[42] Dr. Karen R. Sollins. The TFTP Protocol (Revision 2). RFC 1350, July 1992.

[43] Ahren Studer and Adrian Perrig. The Coremelt Attack. In *European Symposium on Research in Computer Security*, pages 37–52, Saint-Malo, France, 2009. https://doi.org/10.1007/978-3-642-04444-1_3.

[44] Juan Wang, Ru Wen, Jiangqi Li, Fei Yan, Bo Zhao, and Fajiang Yu. Detecting and Mitigating Target Link-Flooding Attacks Using SDN. *IEEE Transactions on dependable and secure computing*, 16:944–956, 2019.

[45] Xinetd. Xinetd Project. https://packages.ubuntu.com/kinetic/xinetd, accessed on October 6, 2023.

[46] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *USENIX Security Symposium*, pages 3865–3881, Virtual Event, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/xing.

[47] Xueyang Xu, Z. Morley Mao, and J. Alex Halderman. Internet Censorship in China: Where Does the Filtering Occur? In *Passive and Active Measurement*, pages 133–142, Berlin, Heidelberg, 2011. https://doi.org/10.1007/978-3-642-19260-9_14.

[48] ZMap. ZMap. https://github.com/zmap/zmap, accessed on October 6, 2023.

Appendices

A Loop Verification Results

Cycle	Loop IPs	Loop Pairs	Success Rate
{454, 535, 454}	71.6 k	24.27 M	96%
{454, 454}	97.8 k	786.88 M	75%
Middlebox Loop (estim.)	74.6 k	0.61 B	(avg) 75.6%
{613, 529, 613}	1.6 k	0.46 M	95%
{613, 455, 613}	6.6 k	6.79 M	70%
{613, 630, 613}	7.3 k	7.76 M	84%
{613, 613}	14.2 k	100.46 M	73%
{613, 914, 613}	15.5 k	8.46 M	80%
{538, 538}	4.3 k	9.08 M	16%
{605, 605}	6.7 k	22.35 M	25%
{455, 185, 455}	6.3 k	1.07 M	91%
{455, 529, 455}	5.0 k	1.76 M	97%
{455, 94, 455}	4.2 k	1.73 M	77%
{455, 455}	10.1 k	50.51 M	51%
{455, 630, 455}	15.0 k	48.25 M	88%
{455, 914, 455}	20.6 k	75.21 M	76%
{296, 296}	4.9 k	12.01 M	24%
{525, 525}	4.9 k	11.95 M	24%
{303, 303}	14.2 k	100.50 M	66%
{529, 914, 529}	15.9 k	5.47 M	83%
{914, 94, 914}	14.6 k	7.73 M	93%
{914, 914}	23.5 k	275.53M	64%
{630, 630}	14.6 k	107.08 M	71%
{185, 185}	15.1 k	113.77 M	63%
{94, 94}	5.2 k	13.52 M	47%
Non-Middlebox Loop (estim.)	36.8 k	0.43 B	(avg) 65.5%
Overall (estim.)	98.4 k	1.01B	(avg) 70.1%

Table 2: DNS Verification Summary

Cycle	Loop IPs	Loop Pairs	Success Rate
{2, 2}	5.0 k	12.40 M	95%
{2, 29, 2}	16.6 k	45.46 M	81%
{48, 48}	0.5 k	0.13 M	100%
{56, 56}	0.6 k	0.17 M	100%
{49, 50, 49}	5.0 k	3.21 M	100%
{49, 49}	4.8 k	11.41 M	98%
{49, 29, 49}	17.0 k	54.09 M	85%
{50, 50}	0.5 k	0.13 M	100%
{29, 29}	13.2 k	87.29 M	82%
Overall (estim.)	19.0 k	0.18 B	(avg) 84.5%

Table 3: TFTP Verfication Summary

Protocol	Loop IPs	Loop Pairs	Success Rate
Active Users	3.2 k	5.24 M	67.5%
Daytime	14.0 k	97.76 M	79.6%
Time	13.2 k	87.05 M	78.8%
Echo	52.7 k	1388.99 M	83.8%
Chargen	22.6 k	254.83 M	40.3%
QOTD	17.9 k	160.68 M	87.8%
Overall (estim.)	96.5 k	1.56 B	(avg) 78.1%

Table 4: Legacy Protocols Verification Summary

B Example Payloads

B.1 DNS

Cluster #296: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=0 z=0 ad=0 cd=0 rcode=format-error qdcount=1 ancount=1 nscount=0 arcount=0 qd=<DNSQR qname=’domain’ qtype=A qclass=IN |> an=<DNSRR rrname=’domain’ type=A rclass=IN ttl=65448 rdlen=4 rdata=ip |> ns=None ar=None |>

Cluster #525: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=1 arcount=0 qd=<DNSQR qname=’domain’ qtype=A qclass=IN |> an=<DNSRR rrname=’domain’ type=A rclass=IN ttl=86400 rdlen=4 rdata=ip |> ns=<DNSRR rrname=’domain’ type=NS rclass=IN ttl=86400 rdlen=29 rdata=’ns_server’ |> ar=None |>

Cluster #529: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=1 arcount=1 qd=<DNSQR qname=’domain’ qtype=A qclass=IN |> an=<DNSRR rrname=’domain’ type=A rclass=IN ttl=86400 rdlen=4 rdata=ip |> ns=<DNSRR rrname=’ns_server’ type=NS rclass=IN ttl=1800 rdlen=29 rdata=’ns_server’ |> ar=<DNSRR rrname=’ns_server’ type=A rclass=IN ttl=1800 rdlen=4 rdata=ip |> |>

Cluster #455: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=0 arcount=0 qd=<DNSQR qname=’domain’ qtype=A qclass=IN |> an=<DNSRR rrname=’domain’ type=A rclass=IN ttl=86400 rdlen=4 rdata=ip |> ns=None ar=None |>

Cluster #630: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=server-failure qdcount=1 ancount=1 nscount=0 arcount=0 qd=<DNSQR qname=’domain’ qtype=A qclass=IN |> an=<DNSRR rrname=’domain’ type=A rclass=IN ttl=86400 rdlen=4 rdata=ip |> ns=None ar=None |>

Cluster #535: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=13 arcount=11 qd=<DNSQR qname=’graph.facebook.com.’ qtype=A qclass=IN |> an=<DNSRR rrname=’graph.facebook.com.’ type=A rclass=IN ttl=61 rdlen=4 rdata=ip |> ns=<DNSRR rrname=’com.’ type=NS rclass=IN ttl=16083 rdlen=20 rdata=’f.gtld-servers.net.’ more DNSRRs |»»»»»> |>

Cluster #454: <DNS txid qr=1 opcode=QUERY aa=0 tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=0 arcount=0 qd=<DNSQR qname=’graph.facebook.com.’ qtype=A qclass=IN |> an=<DNSRR rrname=’graph.facebook.com.’ type=A rclass=IN ttl=60 rdlen=4 rdata=ip |> ns=None ar=None |>

B.2 NTP

Cluster #5696: <NTPPrivate response=1 more=0 version=3 mode=7 auth=0 seq=0 implementation=32 err=incompatible implementation number nb_items=0 mbz=0 data_item_size=0 |>

Cluster #3291: <NTPPrivate response=1 more=0 version=3 mode=7 auth=0 seq=0 implementation=26 err=incompatible implementation number nb_items=0 mbz=0 data_item_size=0 |>

Cluster #6813: <NTPPrivate response=1 more=0 version=3 mode=7 auth=0 seq=0 implementation=10 err=incompatible implementation number nb_items=0 mbz=0 data_item_size=0 |>

Cluster #3072: <NTPPrivate response=1 more=0 version=3 mode=7 auth=0 seq=0 implementation=6 err=incompatible implementation number nb_items=0 mbz=0 data_item_size=0 |>

B.3 TFTP

Cluster #49: <TFTP op=ERROR |<TFTP_ERROR errorcode=Illegal TFTP operation errormsg=’Bad Filename’ |» with single NULL byte ending.

Cluster #29: <TFTP op=ERROR |<TFTP_ERROR errorcode=Not defined errormsg=’Access violation’ |» with single NULL byte ending.

Cluster #2: <TFTP op=0 |<Raw load=’0x0003nameinvalid request’ |»

Cluster #56: <TFTP op=ERROR |<TFTP_ERROR errorcode=Unknown transfer ID errormsg=’Illegal TID’ |» with single NULL byte ending.