A Large-Scale Measurement Study of the PROXY Protocol and its Security Implications

Stijn Pletinckx
University of California,
Santa Barbara
stijn@ucsb.edu

Christopher Kruegel
University of California,
Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California,
Santa Barbara
vigna@ucsb.edu

***Abstract*—Reverse proxy servers play a critical role in optimizing Internet services, offering benefits ranging from load balancing to Denial of Service (DoS) protection. A known shortcoming of such proxies is that the backend server becomes oblivious to the IP address of the client who initiated the connection since all requests are forwarded by the proxy server. For HTTP, this issue is trivially solved by the `X-Forwarded-For` header, which allows the proxy server to pass to the backend server the IP address of the client that originated the request. Unfortunately, no such equivalent exists for many other protocols. To solve this issue, HAProxy created the PROXY protocol, which communicates client information from a proxy server to a backend server at a lower level in the network stack (Layer 4), making it protocol-agnostic.**

In this work, we are the first to study the use of the PROXY protocol at Internet scale and investigate the security impact of its misconfigurations. We launched a measurement study on the full IPv4 address range and found that, over HTTP, more than 170,000 hosts accept PROXY protocol data from arbitrary sources. We demonstrate how to abuse this protocol to bypass on-path proxies (and their protections) and leak sensitive information from backend infrastructures. We discovered over 10,000 servers that are vulnerable to an access bypass, triggered by injecting a (spoofed) PROXY protocol header. Using this technique, we obtained access to over 500 internal servers providing control over IoT monitoring platforms and smart home automation devices, allowing us to, for example, regulate remote controlled window blinds or control security cameras and alarm systems. Beyond HTTP, we demonstrate how the PROXY protocol can be used to turn over 350 SMTP servers into open relays, enabling an attacker to send arbitrary emails from any email address. In sum, our study exposes how PROXY protocol misconfigurations lead to severe security issues that affect multiple protocols prominently used in the wild.

I. Introduction

With the increase of users and devices connected to the Internet, scaling Internet services has become a crucial part of modern-day infrastructure hosting. One pivotal component that enables many of today's scaling solutions is the reverse proxy server [7]. In general, proxy servers reside on the path between a client and one or more backend servers and forward the incoming requests from each client to the backend infrastructure [9]. Reverse proxies, in particular, are commonly used for load-balancing incoming client connections across a group of server backends. This allows a system administrator to dynamically add and remove backend servers, according to current demands, without the need to change any frontend infrastructure. A reverse proxy can either be hosted in-house or outsourced to a third-party provider, such as Cloudflare [8] or Akamai [1].

One downside of proxy-level load balancing is that each incoming request will appear to the backend server as if it was initiated by the reverse proxy itself. This inhibits the backend server from correctly logging page retrievals, as well as doing any form of access control or blocklisting at the host level. In HTTP, this shortcoming is addressed by including an `X-Forwarded-For` header in the request from the proxy server to the backend server. By setting the value of this header to the IP address of the client who initiated the request, the backend server becomes aware of the actual host visiting the page.

Unfortunately, most other protocols do not have an equivalent of the `X-Forwarded-For` HTTP header. To solve this, proxy software developer HAProxy created the PROXY protocol in 2010 [32]. The core idea of the PROXY protocol is to create a header that the proxy server communicates to the backend server shortly after finishing the TCP handshake. This header contains all information that the backend server would have received if the client had directly connected to the backend server instead of going through the proxy server (e.g., the client's source IP address and source port). By parsing the header, the backend server can then use this information in access logs, for access control, or for other use cases (see Section II for more details about the PROXY protocol). To avoid abuse of this protocol, HAProxy advises that each backend server maintains a list of trusted proxy servers, and only accepts and parses PROXY headers stemming from these trusted proxy servers. If not, an adversary could spoof the content of the PROXY header (such as the source IP address), and potentially poison the server logs and bypass access control mechanisms.

In this paper, we perform the first large-scale measurement study of PROXY protocol usage in the wild and identify several misconfigurations that lead to severe security issues. We demonstrate techniques that abuse these misconfigurations to bypass on-path proxies, therefore circumventing proxy-based security mechanisms (e.g., Denial-of-Service (DoS) protection). Furthermore, we show that backend servers actually

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242247
www.ndss-symposium.org

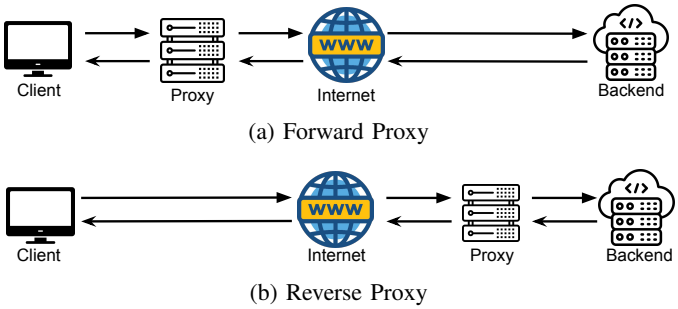
rely on PROXY header information for access control, and find hundreds of servers in the wild vulnerable to PROXY header spoofing attacks (see Section IV-D for details on the ethical considerations taken for this research). In the case of SMTP, we could use the PROXY protocol to turn more than 350 email servers into open relays, which allows adversaries to spoof arbitrary email addresses without any form of authentication. Generally, open email relay servers are easily detected and added to blocklists. However, when requiring the PROXY protocol, these open relays remain undetected, as current scanners do not use this protocol when looking for open relays in the wild. As such, the open relay servers found during our study are not only harmful but also persistent. In summary, this paper makes the following contributions:

- We launch the first large-scale measurement study on PROXY protocol usage within the entire IPv4 space and find that a large number of hosts accept PROXY header information from arbitrary sources (177,983 over HTTP, 2,332,377 over SMTP, and 2,343,420 over SSH).
- We find over 10,000 instances where PROXY header injection in HTTP connections leads to access control bypass. Beyond HTTP, we also show how we bypass on-path proxies (and therefore proxy-based security mechanisms) over SMTP (25,366 instances) and SSH (30,882 instances).
- We demonstrate how the PROXY protocol can be used to expose internal hosts. For example, we find 268 instances over SMTP that provide a `.home` TLD, which is exclusively used within internal infrastructures.
- Using the PROXY protocol, we obtain access to over 500 hosts that provide control over home automation devices, IoT monitoring platforms, or other types of sensitive dashboards and devices, allowing us to regulate, for example, remote controlled window blinds or control security cameras and alarm systems.
- Using the PROXY protocol, we turn 373 SMTP servers into open relays, giving us the ability to send arbitrary emails from any email address.

During our measurements, we found hundreds of vulnerable servers, ranging from educational institutions and security companies, to hosting providers and Internet Service Providers (ISPs), showing that this vulnerability exists on (supposedly) well-maintained infrastructures. We notified all affected parties through responsible disclosure whenever possible. We provide details of our responsible disclosure in Section VIII.

II. Background

A classic design pattern adopted by many Internet services today is the "client-server" model. The idea is that a client connects to a server (often referred to as the backend server) that hosts a service and allows the client to interact with that service. Indeed, while the number of servers can be relatively few, the number of clients can potentially be many. To efficiently scale and protect this model, we often find an additional, yet generally hidden, component implemented in-between the client and backend server, called a *proxy server*.



(a) Forward Proxy

(b) Reverse Proxy

Fig. 1: Proxy server types. Figure (a) shows an example of a *forward proxy*, which resides commonly between the client and the Internet backbone. Figure (b) shows an example of a *reverse proxy*, which resides commonly between the Internet backbone and the backend server(s).

A. Proxy Servers

The goal of a proxy server is to improve the performance, security, and/or privacy of an Internet application. In HTTP, for example, static web pages can be cached by a proxy server. When a client requests a cached web page, the proxy can serve the page directly as opposed to first contacting the backend server. This reduces the load on the backend infrastructure and improves the response time for a request. Furthermore, proxy servers can check requests and responses for any potential malicious activity (i.e., JavaScript injection to perform cross-site scripting), and subsequently abort the communication when needed [35].

From a backend server's point of view, any request involving a proxy will look as if it was initiated by the proxy itself. This makes the backend server oblivious to the original client's IP address, therefore increasing the privacy of the client. This may not always be desired by the backend application, which may need this information for logging purposes or access control decisions. As such, in HTTP, the proxy server can set the `X-Forwarded-For` header to communicate to the backend server information associated with the client.

A proxy server can reside in two locations on the network path: either between the client and the Internet backbone, in which case we call it a *forward proxy*, or between the Internet backbone and the server, referred to as a *reverse proxy*. We elaborate below.

Forward Proxies: In a forward proxy setting, the proxy is responsible for forwarding the client's packets to the Internet, until they eventually reach their intended server. Figure 1a depicts a representation of such a configuration. Because forward proxies are purposefully used by the client, their goal is often to hide the client's IP address (since from the server's side, it looks as if the initial packet was sent by the proxy), and/or access hosts that would otherwise not be available to the client.

Another use case of these proxies is for organizations to block and monitor outbound access from within a company network. For example, a high school could implement a forward proxy for the student's network such that it can restrict access to social media or gaming servers.

Reverse Proxies: In a reverse proxy setting, the proxy resides closer to the backend server, often sharing the same internal network (see Figure 1b). This proxy type is regularly used as a load balancer to dynamically allocate requests within the backend infrastructure if, for example, a web page is hosted on multiple servers. Since all incoming requests have to flow first through the proxy, the backend servers can be hidden behind a firewall, making them unavailable to the public for direct attacks. Then, at the proxy level, access control and filtering can be applied to prevent malicious requests from reaching the internal network. This creates several layers of security, while providing the opportunity to scale, allowing for a more robust and modular infrastructure.

Contrary to a forward proxy, a reverse proxy is usually under the control of the same organization that runs the backend servers, through either an in-house solution or by renting a proxy server from a third party. As mentioned before, the `X-Forwarded-For` HTTP header allows the proxy to communicate the IP address of the client that initiated the request to the server. However, a fundamental shortcoming of this approach is that it exclusively works for HTTP(S). As such, backend servers need to look for different solutions if they want to log similar information from protocols other than HTTP. To solve this problem, HAProxy created the PROXY protocol [32].

B. HAProxy PROXY Protocol

The HAProxy PROXY protocol aims to transport client information from the proxy server to the backend server, regardless of which application-level protocol is being used [32]. The idea is to insert a header (called the PROXY protocol header) at the start of each connection between the proxy and the backend server. This header carries over client information, such as the source IP address and the source port, which can be accessed and used by the backend server. At the moment, two versions are specified (Version 1 and Version 2), which we explain in more detail below.

Version 1 – Plaintext Header: Version 1 of the PROXY protocol uses a plaintext header, shown in Listing 1. The header prefix starts with the keyword `PROXY`, followed by the IP protocol version, which can either be `TCP4` for IPv4, or `TCP6` for IPv6. The next 4 fields are reserved for the source IP address of the client that initiated the request, the source port of that same client, the destination IP address of the backend server, and the destination port of the backend server. The header terminates with a CRLF sequence.

```
PROXY TCP4 1.2.3.4 1337 4.5.6.7 80
```

Listing 1: Example of a PROXY protocol header. The header starts with the "PROXY" indicator, followed by the IP protocol version. Then, the header contains the source IP address, source port, destination IP address, and destination port.

The simplicity of the v1 PROXY protocol header makes it easy for existing server software to implement supporting functionality, both from the sender side (as a proxy) as well as on the receiver side (as a backend server). The plaintext header is intentionally designed to not be mistaken for a valid HTTP request or other protocol-specific formats. Even if no support is available to parse a PROXY protocol header, the format will not get misinterpreted, but will rather yield an error on the server side. Therefore, administrators do not have to worry about receiving accidental PROXY headers.

In practice, when the proxy server receives a request from the client, the proxy server will create a PROXY protocol header using the IP address and port number of both the client and the server. Prior to forwarding the request of the client to the backend server, the proxy server will include the created header during the initiation of the connection with the backend server. The backend server can then parse the information in the header, and use it accordingly.

Version 2 – Binary Header: Version 2 of the PROXY protocol adopts a binary format. Although harder to debug, the binary header is more efficient to parse. Especially for IPv6, there are various formats to represent an address, which are harder to parse in plaintext compared to binary. Currently, Version 2 of the PROXY protocol is not yet supported by most server applications. Given the fairly recent advent of this new protocol, the Version 1 header was intentionally designed for a smooth roll-out that is easy to debug. Many server-side applications are, therefore, still limited to only providing support for Version 1 of the PROXY protocol. As such, we focus the rest of this study on the Version 1 header of the PROXY protocol.

III. PROXY Protocol Security Considerations

The HAProxy PROXY protocol header is low in complexity, making it straightforward to parse for servers. Correspondingly, for a sender, it is relatively easy to create such a header and incorporate it during a connection setup. However, from an adversarial point of view, it is equally easy to temper with the PROXY header, which can potentially lead to unwanted behavior. To prevent this, HAProxy recommends in its specification to only accept PROXY headers from a set of known sources [32]. This means that a backend server should never attempt to detect whether a PROXY header is present, but should rather know a priori whether the request will be received from a proxy server that will include the PROXY protocol implementation. HAProxy does not provide any detail on how or where this distinction should be made, i.e., whether this should be incorporated into existing server software (e.g., using a lookup table for the source IP), or at the firewall level. To get an overview of how this happens in practice, we evaluate the two most popular server software at the time of writing: NGINX and Apache.

A. Simulation Environment

We simulate a proxy environment on a local setup to study the PROXY protocol in further detail. We implement two backend servers (one using NGINX, one using Apache) that reside behind a proxy server (running HAProxy) that loadbalances among the two backend servers (see Appendix A for more details).

For both backend servers, we use the standard off-the-shelf configuration file that gets pre-installed during setup, which runs an HTTP server on port 80. We then configure the load balancer to forward all web traffic to both backends in a round-robin fashion. When requesting the web page, our communication first goes through the proxy server, after which

the proxy server chooses one of the backend servers to forward the request to[1]. Upon examining the logs of both backend servers, each request is attributed to the proxy server, and no information is recorded about the initiating client.

To enable the PROXY protocol on our load balancer, we first configure HAProxy to add the PROXY header when initiating a request to any of the backend servers. We do so by adding the `send-proxy` keyword in the configuration file as shown in Listing 3 of Appendix A. Subsequently, we configure both backend servers to parse the PROXY protocol during a connection setup.

PROXY Protocol Configuration: Both NGINX and Apache make it trivial to enable PROXY protocol support on the backend side. For NGINX, we merely append `proxy_protocol` to the `listen` instruction (Listing 4 in Appendix A), and for Apache, we add the line `RemoteIPProxyProtocol On` to our virtual host (Listing 5 in Appendix A) and include the `remoteip` module. While both configurations have a common simplicity, the effects are surprisingly different between the two systems. Both will correctly parse any incoming PROXY header, but where NGINX will ignore any information from the header, Apache by default will change the value of the incoming IP address from the proxy's IP address to the IP address mentioned as the source IP address in the PROXY header (i.e., the IP address of the client who initiated the request).

Recall from Section II that the PROXY protocol specification instructs to configure a list of known hosts from which to accept and parse the PROXY header. At the moment of writing, both NGINX and Apache do not have any mechanism in place to do so. Moreover, neither of the software vendors mention anything about this precaution in their documentation of the PROXY protocol. It therefore requires a proactive system administrator to create and implement a design for handling this functionality that is both efficient and easy to maintain. If not, any client can send a PROXY header directly to the backend server. Moreover, given Apache's default behavior of changing the source IP address of any incoming request to the one specified in the PROXY header, an adversary can easily spoof the content of the PROXY header, which will be parsed and accepted by the backend server.

B. Security Implications

Based on the implementation described above, we identify 3 misconfigurations that can be abused using the PROXY protocol: 1) The backend infrastructure is not hidden from the public, for example, using a firewall. This allows any client to directly connect to the backend server, circumventing the proxy server. 2) The backend server does not check whether a connection stems from a trusted and known proxy. 3) The backend server does not validate whether the client originating the request has access to the requested content (and potentially assumes that the proxy server already took care of this step).

Figure 2 shows an overview of how an attacker can abuse the PROXY protocol on misconfigured servers. In scenario A, we depict how an authorized client legitimately interacts with a server using the PROXY protocol. Instead of directly connecting to the backend server, the client connects to the proxy server, which then forwards the request prefixed by a PROXY header to the backend server. In scenario B, we depict how this setup prevents an unauthorized attacker from accessing restricted content. If the attacker connects to the proxy server using an HTTP `GET` request, the proxy recognizes that the IP is not authorized, and denies the request. If the attacker instead connects to the backend server directly, using an HTTP `GET` request, the backend server will discard the request, since it is not adhering to the PROXY protocol format.

However, if the attacker directly connects to the backend server and manually injects a PROXY header (scenario C), the backend server might believe that the connection stems from a proxy server, and hence, operates under the assumption that access control checks were already performed by the proxy. Furthermore, many proxy servers provide Denial of Service (DoS) protection by rate limiting or blocking abusive behavior of clients. By using the PROXY protocol to directly connect to the backend server, an adversary bypasses the proxy, and hence also bypasses any proxy-based protections. For scenario C, the adversary abuses all three misconfigurations described above.

Nonetheless, even if the backend server checks whether the initiating client has access to the requested content (therefore eliminating the third misconfiguration), an adversary can still spoof the source IP address value in the PROXY header. This is depicted in scenario D. If the adversary knows which IP address range is accepted by the backend, they can spoof a value within that range. If not, the adversary will have to guess an accepted IP address. However, we will later show in our experiments that choosing an internal network address (such as for example `10.0.0.10` or `localhost`) is often sufficient to bypass this access control check.

We tested scenario D in our simulation environment. We configure our Apache server to only allow requests stemming from the IP `192.168.56.100`. When sending a regular HTTP `GET /` request (i.e., without PROXY header) from an IP address different than `192.168.56.100`, the server returns a `403 Forbidden`, as expected. However, if we send from the same client an HTTP `GET /` request with a PROXY header that has a source IP address value of `192.168.56.100`, we receive back a `200 OK`, indicating we successfully bypassed the access control mechanism. In addition to bypassing DoS protection and bypassing access control checks, injecting a PROXY header can also lead to "log forging" [28], meaning that an attacker can poison the logs of the backend server.

Note that the above mentioned vulnerabilities are not due to a fundamental issue in the PROXY protocol itself, but rather due to a misconfiguration on the backend server. By default, the backend server will trust PROXY protocol information from any source that sends it. If an administrator is not aware of these default settings (and hence, does not change them), the backend server runs in a misconfigured setting, which harms the security posture of the backend infrastructure. In our study, we explore these misconfigurations in the wild and find that hundreds of hosts are vulnerable to the above described attacks.

[1]Note that, for simplicity, we configured no caching mechanism.

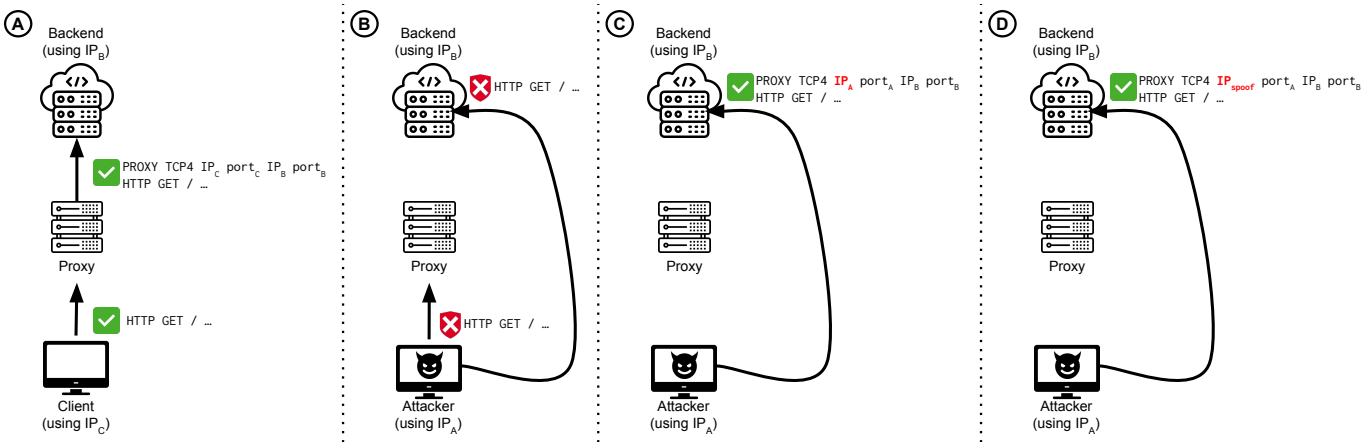


Fig. 2: Attack scenario using the PROXY protocol. Scenario A shows a legitimate (and typical) interaction between the client and the infrastructure. Scenario B depicts how an attacker is not able to access both the proxy and the backend server using regular HTTP request. However, by injecting a PROXY header (depicted in Scenario C) the attacker can directly communicate to the backend server and bypass the proxy server. Scenario D shows an advanced version of this approach where the attacker also spoofs the source IP address value in the PROXY header to circumvent access control checks on the backend side.

C. HTTP `X-Forwarded-For` Header

For HTTP, the goal of the PROXY protocol is equal to that of the `X-Forwarded-For` HTTP header: as a proxy, provide information to the backend server that it would have otherwise also received if there was no proxy in place. It is therefore reasonable to assume that the above security issues are equally present in the implementation of the `X-Forwarded-For` header. We test this in our simulation environment and find that there are actually much more robust parameters available for the `X-Forwarded-For` header, specifically aimed at improving security. For example, Apache allows for a `RemoteIPTrustedProxy` option, which sets a list of proxies trusted by the backend server. If a request contains an `X-Forwarded-For` header but does not stem from a trusted proxy, the value in the header will not be used by the backend server. Such functionality is missing from the PROXY protocol implementation in Apache.

IV. Methodology and Ethics

We study the prevalence and security impact of the PROXY protocol through a large-scale measurement study on the Internet. Performing Internet-scale measurements requires careful planning and considerations to ensure sound data collection while remaining within the bounds of ethical research. Below we motivate our methodology and scope, and highlight our threat model. We also discuss the ethical considerations for this study.

A. Aim and Scope

The goal of this study is to capture potential PROXY protocol misconfigurations in the wild, and assess the security posture of the corresponding infrastructure. Since there exists no robust methodology to detect proxied services in the wild and at scale, we have to cast a wide net and rely on Internet scanners to find potential hosts residing behind a proxy. The idea is to find instances that accept the PROXY header. We know from Section III that backend servers receiving connections with the PROXY protocol header should only consider incoming requests as valid when they originate from a known source (i.e., the designated proxy server). As such, if we probe a server using the PROXY header and receive back a successful `200 OK` response, we have a potential misconfiguration, given that our scanning infrastructure is (to the best of our knowledge) not part of any backend server's list of known proxies.

We aim to not only shed light on the prevalence of these misconfigurations but to also demonstrate their security impact, and explain what can be done from a website administrator's side, as well as from server-software developers' and the HAProxy PROXY protocol maintainers' perspectives.

HAProxy maintains a list of server software distributions that currently offer support for the PROXY protocol [33]. Most dominantly present in this list is software meant for serving the HTTP protocol. As such, we focus the majority of our work on HTTP (Section V). Nonetheless, HAProxy also lists several SMTP-supporting software. We therefore launch a second study focusing on the SMTP protocol (Section VI). Lastly, industry players such as Cloudflare offer support for interpreting the PROXY protocol through third-party software. For example, `mmproxy` is a TCP proxy that resides near the application, and translates connections coming from the loadbalancer and that contains the PROXY header [6]. It does so by creating a new packet (without the PROXY header) and spoofing the IP address and port number in the newly created packet according to the values from the initial PROXY header. This new packet then gets forwarded to the application, which will interpret the packet as if coming directly from the originating client. Given the existence of such functionality, we also take a look at a protocol not listed by HAProxy, namely SSH, and assess whether it makes use of the PROXY protocol in the wild (Section VII).

B. Data Collection

To find instances of servers accepting the PROXY protocol, we first need to know which servers operate our applications of interest. Using ZMap, we initiate each experiment with an Internet-wide port scan to collect all hosts listening to a specific port (e.g., for HTTP, this would be port 80). ZMap is a stateless scanner that efficiently scans the entire IPv4 address range at the Layer 4 level (using TCP `SYN` packets). Next, we feed the results of ZMap into ZGrab, which is a more fine-grained Internet scanner that operates at Layer 7. Because ZGrab does not support the PROXY protocol by default, we develop supporting functionality for it in the `http`, `smtp`, and `ssh` modules of the scanner.

For each protocol, we launch six different tests on each target IP address gathered from the ZMap scan. Each test is designed to observe the potential differences in the behavior of the server when changing the values in a PROXY header. Our first test functions as a baseline, and consists of a regular probe without an injected PROXY header. For the second test, we inject into each packet a PROXY header that contains the IP address of our scanning machine in the source IP address field of the header. This gives us an initial idea of how servers react when receiving a PROXY header from an arbitrary source on the Internet[2]. In case the baseline probe is unsuccessful, but we receive a status code of `200 OK` with the injected header, we assume that we potentially bypassed the on-path proxy (see scenario C in Figure 2). If this attempt is unsuccessful, we try again using a spoofed value for the source IP address in the PROXY header. Specifically, we launch four tests, each with a different IP address in the PROXY header, representing an internal address from the following list:

- `127.0.0.1`
- `10.0.0.10`
- `172.16.0.10`
- `192.168.0.10`

By using the localhost loopback address along with three common internal network addresses [25], we test if the receiving server believes that the initial request originated from within the server's network. We chose these IP addresses to potentially circumvent access control checks by the backend server on the source IP address of the initiating client (scenario D in Figure 2). Recall from Section III that an adversary needs to spoof a known IP address to the backend server to bypass the backend server's access control. To avoid having to bruteforce known IP addresses, we hypothesize that internal network addresses will generally be granted access to content within the infrastructure. If a target server returns an unsuccessful response to the first two probes, but responds successfully to a probe with a spoofed IP address, we have potentially bypassed the access control checks of the server using the PROXY protocol. To verify each bypass, we launch a final probe to confirm that the server granted this bypass because of the PROXY protocol and not because it received unexpected data [19]. Concretely, we send a request with a malformed PROXY header, shown in Listing 2, that contains "ABC" as the protocol, which is invalid according to the standard. If this time we receive an unsuccessful response, we confirm that the server indeed reacted to the PROXY protocol. Figure 3 visualizes our methodology.

[2]Note that if the server does not support the PROXY protocol, it will simply return an error when receiving a PROXY header. See Section II for more details.

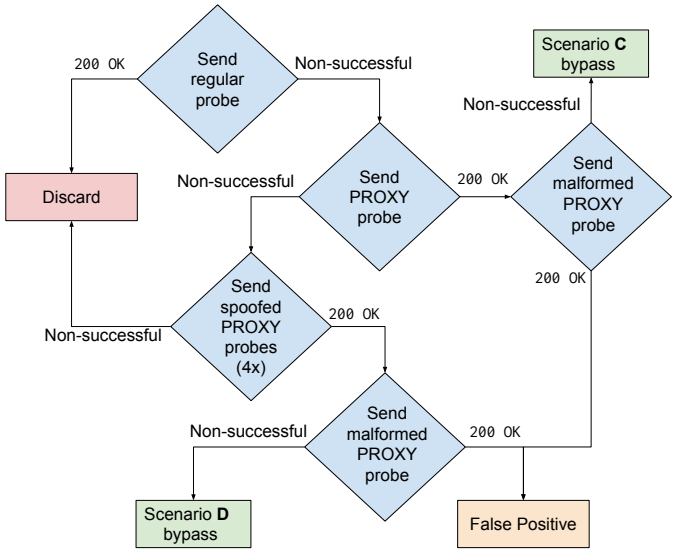


Fig. 3: Flow diagram of our measurement methodology. We first find all instances that do not give us a successful response with a regular probe. Then, we probe using a PROXY header containing our own IP address. If we do get a successful response to this probe, we achieved the bypass of scenario C in Figure 2. If unsuccessful, we try again using a spoofed source IP address in the PROXY header to see if we can achieve a bypass of scenario D. For each successful bypass, we check whether the target server indeed reacts based on the PROXY header by sending a malformed header. If, however, the target also responds successfully to this, we consider the bypass a false positive.

PROXY ABC IP_{src} $Port_{src}$ IP_{dst} $Port_{dst}$

Listing 2: Malformed PROXY protocol header used for sanity check. Notice that the second field shows "ABC" rather than a known protocol, such as "TCP4"

Since our four spoofed tests have the potential to bypass access control, we apply an abundance of caution to avoid leaking any personally identifiable information (PII), or other (security) sensitive content. Concretely, for HTTP, we only perform a full HTTP `GET` request for the non-PROXY header and the PROXY header containing our own IP address, but resort to an HTTP `HEAD` request for the scans containing an internal network IP address. This avoids accidentally receiving an internal web page containing PII. We elaborate more on our safeguards and ethical considerations in Section IV-D.

C. Threat Model

For this study, we assume an attacker external to the network, with no prior knowledge of the network infrastructure or access control policies. They use ordinary hardware on an average Internet connection, having no access to specialized measurement setups or large-scale server pools. We assume

the attacker wants to bypass a network's access control mechanisms. The target network runs one or more backend servers, which are configured behind a reverse proxy. All backend servers expect a PROXY header for each connection. We assume access control is either implemented at the backend level or at the proxy level.

D. Ethics

Research involving active measurement probes requires careful evaluation of the potential ethical implications [14]. We use ZMap for our initial port scan, which has a known fingerprint (IP ID of 54321) that is easy to block for system administrators [13]. As for the ZGrab scan, we follow best-practices [13], [2] by providing several opt-out channels:

1) For HTTP, all scans contain a custom User-Agent with a description of the study and a contact email.
2) The server from which we scan can be reached over HTTP(S), and redirects to a website explaining our research and providing contact details.
3) We configure a TXT record for the server from which we scan that contains contact info and a reference to our website.

In total, we received two opt-out requests.

As mentioned before, for the HTTP requests aimed at demonstrating a potential access bypass, we refrain from doing a full HTTP `GET` request, and only collect the status code received from performing an HTTP `HEAD` request to avoid collecting PII. Furthermore, at no point in our experiment do we escalate any of our obtained privileges, or try to scrape sensitive information from any of the collected web pages. Our research study follows the guidance provided in the Menlo report [2]. We always evaluate the utility of our research against the potential harm it may cause. Finally, we communicate all discovered security issues to the involved parties whenever possible (see Section VIII for our full responsible disclosure process).

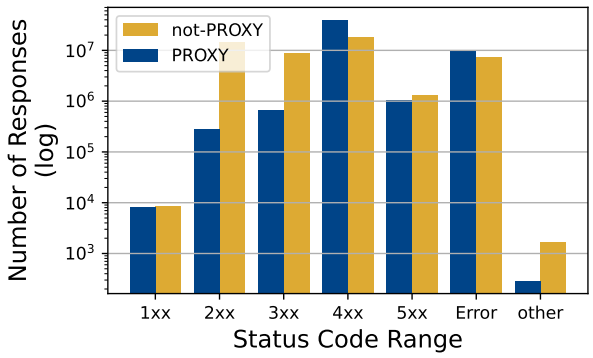
Later in this paper we show how including the PROXY header can give access to sensitive web pages and platforms, such as smart-home control systems and facility monitoring services. At no point in this study do we execute commands on any of these platforms. Lastly, we restrict access to the collected data to those involved with this study.

V. HTTP Results

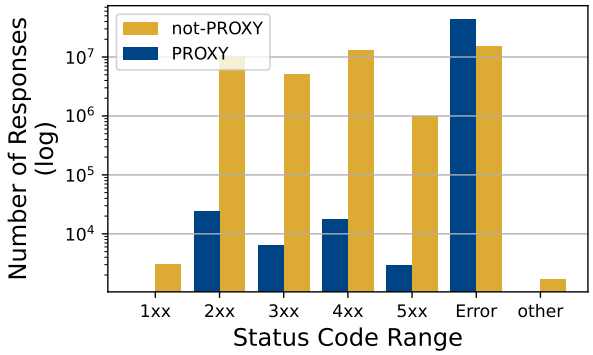
In this section, we present the results of our large-scale measurement study on the HTTP protocol (Sections VI and VII cover SMTP and SSH, respectively). Apart from gaining general insight into host responses to the PROXY protocol, we expose misconfigurations of this protocol in the wild, which in some cases lead to information leakage, infrastructure exposure, or access control bypass.

A. General Observations

From our ZMap scan, we collect 51,246,816 hosts listening on port 80 and 44,752,582 listening on port 443. After feeding this information into our ZGrab scan, we obtain more details



(a) HTTP



(b) HTTPS

Fig. 4: Status code responses received during our measurements. We only report on the responses for the requests sent without a PROXY header, and the requests sent with a PROXY header containing our real IP address in the source address field. Figure (a) depicts the results for HTTP and Figure (b) for HTTPS. Note how for HTTP, the variety in status codes is rather high, which should not be the case according to the PROXY protocol specifications.

about the responsiveness of these hosts. Figure 4 shows a summary of the HTTP status codes received. Note that, for now, we focus only on the probes without a PROXY header and the probes with our server IP address in the PROXY header. What stands out immediately is the discrepancy between HTTP and HTTPS. For HTTPS, the majority of the responses are as expected, and in line with the HAProxy specifications of the protocol, namely: respond with an error to each PROXY header received from a host not in the known-proxies list, or if the protocol is not supported. For HTTP, on the other hand, the responses are much more dispersed among the ranges of status codes, creating an interesting spectrum for further inspection. As such, we focus the rest of our results on the HTTP set.

While we do see that a majority of the responses to our PROXY header returns an error or a response in the `4xx` range, there is still a significant number of responses that return a status code in the `2xx` range (280,713), indicating we successfully retrieved content from the server. However, even with a `2xx` status code, a response can still return an error page. As such, we filter out bogus success responses, which

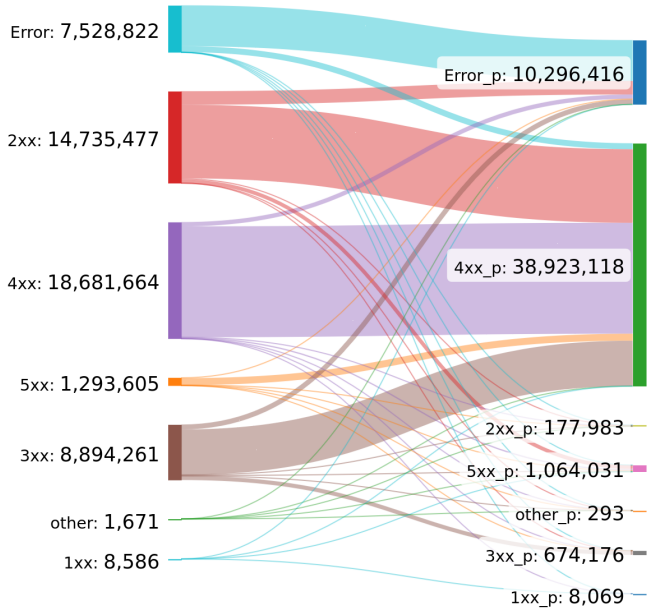


Fig. 5: The change in response code after using a PROXY header (containing our real IP address). On the left side, we see the number of responses received for each status code range when sending a request without a PROXY header to each host. On the right, we see the same but when sending a PROXY header with our IP address in the source address field. The flow shows the change in status code per host. Notice that some of the `2xx` responses on the right side stem from `4xx` or other error responses from the left.

are pages that have an HTTP status code of `200 (ok)`, but display a text similar to "Page Not Found," "Access Denied," "Domain Suspended," etc. We do this by creating a list of common phrases and patterns that occur on such pages and discarding each web page that matches an entry in our list. In total, we collected 63 such patterns[3]. This filters out 102,730 web pages or 36.6%.

Figure 5 provides more detail on how our target hosts respond differently to requests with and without a PROXY header. The left-hand side of the figure depicts the status code ranges received during our regular scan (i.e., using no PROXY header), while the right-hand side depicts the responses received after sending a PROXY header with our own IP address. From this Sankey plot, we see that the majority of servers responding with a `2xx` status code to the PROXY header gave a similar response if given a regular request without the header. We are more interested in the cases that initially would not respond within the `2xx` range, yet do so when presented with a PROXY header. We investigate these hosts in more detail later in the section.

[3]While we initially started with a smaller number of patterns, by investigating various pages throughout our study we encountered more patterns that were not present on our list. These patterns were each time added, after which we ran our analysis pipeline again. Eventually, we reached 63 patterns.

TABLE I: Classification results. We noticed a significantly higher amount of boilerplate pages and login pages for our experiment group compared to our control group. Note that the "Miscellaneous" label indicates the number of pages that could not be classified into any of the other groups.

	Experiment Group	Control Group
Boilerplate	118,641 (66.66%)	64,544 (36.83%)
Login	34,474 (19.37%)	12,136 (6.82%)
Miscellaneous	12,791 (7.19%)	75,897 (42.64%)
Enable JavaScript	6,577 (3.70%)	1,439 (0.81%)
Cannot parse	3,697 (2.08%)	3,696 (2.08%)
NGINX	1,345 (0.76%)	12,875 (7.23%)
Temperature Measurement	304 (0.17%)	6 (<0.01%)
Construction	107 (0.06%)	507 (0.28%)
Apache	47 (0.03%)	6,883 (3.87%)

B. *Source Code Clustering and Analysis*

Recall from Section III that, according to the specifications, a backend server should only act on a PROXY header when the connection comes from a known proxy. Yet, our measurements show that over 170,000 hosts return an HTTP status code in the `2xx` range when sending a PROXY header from our server, which does not function as a proxy to any server on the Internet. As such, we investigate these pages further.

We cluster together all web pages that are near-duplicates of each other. We do this by creating a fingerprint for each web page, using Charikar's fingerprinting algorithm [4], and clustering together pages with the same fingerprint. We chose Charikar for its small memory footprint and fast performance [17], [21]. This creates 9,429 clusters, with the biggest cluster containing 31,284 pages.

We manually inspect a set of pages from the biggest clusters and classify them based on common patterns. Concretely, we take one web page from each cluster with a size of 100 or more. Since each cluster consists of (nearly) identical web pages, we only have to inspect one page per cluster. In total, we inspect 60 clusters. Table I shows the result of our classification. To compare, we also do the same for a control group of the same size as our experiment group, consisting of web pages that responded with a code in the `2xx` range during our initial scan (i.e., without adding any PROXY header).

The majority of the web pages in our experiment group (66.66%) contain what we label as "boilerplate" code: an HTML file that contains a template for a web page but does not have any content in place (yet). We assume these were put online with the intent for further development, but have not been finished. Previous work has shown that this is indeed a common sight on the Internet, and has been specifically observed within the context of orphaned web pages [29]. The number we observe is noticeably larger than the amount of boilerplate web pages in our control group (66.66% vs. 36.83%). This suggests that the pages we found are not meant to be online (yet), and potentially still under development.

The second most encountered type of web page in our experiment group are login pages (19.37%). Again, we notice a stark difference compared to our control group where only 6.82% of all web pages are login pages. This could indicate

that these pages are internal (potentially residing behind a proxy) and hence require login credentials for authorized access. Moreover, if the proxy server protects these pages from brute force attempts or injection attacks [35], an attacker can circumvent this using the PROXY protocol. Due to ethical reasons, we did not make an attempt to verify this hypothesis.

Interestingly, 304 (0.17%) web pages display a web portal for monitoring temperature measurement sensors in locations such as data centers or power plants. In our control group, we find an insignificant number of these web pages (less than 0.01%) demonstrating again that these pages are (unintentionally) exposed via the PROXY protocol. We later demonstrate how we find more of these sensitive web pages in our experiment group.

The remainder of our categories are pages displaying an error message saying we need to enable JavaScript (3.70%), NGINX or Apache default pages (0.76% and 0.03%, respectively), or pages noting that the web page is "under construction" (0.06%), which is similar to the boilerplate code we classified earlier. Notably, we discover that the number of default NGINX and Apache pages is noticeably higher in our control group (7.23% and 3.87%, respectively).

Miscellaneous Web Pages: For both groups, we are still left with a "Miscellaneous" category, which contains the pages we could not classify in either of the other categories. We highlight that, for our control group, this is the category with the largest number of web pages (42.64%). This shows there is a high variety among the web pages in our control group, which is to be expected since these pages are meant to be "regular" and publicly accessible. We verify this by manually inspecting a sample of 200 web pages. In this sample, we indeed only observe common web pages such as company websites, product advertisements, event information, personal websites, and educational institutions.

For our experiment group, the miscellaneous category is considered the long tail of our classification (7.19%). We again sample 200 web pages for manual inspection. Contrary to our control group, we barely encounter any regular web pages in this category. Rather, we notice that the majority of the web pages in this category provide an access portal to home automation systems, temperature sensors, electric vehicle charging station diagnostics, Internet of Things (IoT) sensors, and intrusion alarm monitoring, among others. Concretely, we find that 36% of the inspected web pages are instances of such portals, all of which provide us with some sort of platform to execute commands. For example, in the case of the home automation portals, we could potentially open and close blinds, control light switches, or regulate building climate. For ethical reasons, we never executed any command on any of the encountered web pages.

Several monitoring portals used the same source code for displaying data, suggesting the presence of a general platform for displaying data stemming from sensors or external APIs. We searched the remainder of the miscellaneous group for this particular platform and found a total of 556 hosts running this monitoring software, exposing sensitive information such as network configurations, temperature measurements, water levels, processor usage, memory usage, and other sensor data. These servers are hosted in 37 different countries, with the majority residing in the US (57.01%), followed by Spain (7.91%). They are spread across 67 different Autonomous Systems (ASs), the majority being under the control of Verizon or Vodafone Spain.

The remainder of the pages we inspect manually are misclassified error pages, misclassified login pages, or regular pages. Note that we merely investigated a sample of the web pages, **revealing that 36% expose sensitive access and data.** Given that the total size of our Miscellaneous category spans 12,791 pages, we could **potentially extend this result to over 4,600 pages exposing sensitive information.** This analysis shows that misconfigured proxy infrastructures can expose sensitive portals. As such, deploying the PROXY protocol can provide a false sense of security, since the web pages behind them appear to be inaccessible, yet are uncovered by injecting the PROXY header. Moreover, this incentivizes an adversary to incorporate the PROXY header into their target reconnaissance, since our study shows that scanning the Internet for this protocol offers a subset of sensitive pages at a volume that is easy to process for targeted attacks. We further discuss these implications in Section IX.

From our final probe (using the malformed PROXY header) we received zero successful responses, indicating that we achieved access to the sensitive platforms due to our PROXY header injection. Notably, when we try the same approach using the `X-Forwarded-For` HTTP header field, we only retrieve 9 successful responses, showing that this phenomenon is almost fully unique to the PROXY protocol.

C. Access Bypass

Recall from Section III that some servers perform access control checks on the source IP address given by the PROXY header. In scenario D of Figure 2 we explain how this can still lead to an access bypass if the attacker spoofs the source IP value of the PROXY header. In our measurement experiment, we therefore included several probes with a different source IP address in the PROXY header to detect potential access bypasses in the wild (see Section IV for more details). Concretely, we take each host that responds unsuccessfully to our baseline probe, yet responds with a status code in the `2xx` range when including a (spoofed) PROXY header, indicating an access bypass. We consider a response unsuccessful if it comes with a status code in the `4xx` or `5xx` range, or if we receive a connection error.

Table II summarizes our results. For each test, we report on two numbers: the initial results (written in parentheses), and the results after filtering out the false positives. We consider a result a false positive if it also responds successfully to a malformed PROXY header. Considering all probes, we find that **including a PROXY header leads to an access bypass on 10,089 hosts.** In 5,863 cases, it does not matter which source IP address we provide in the PROXY header. Nonetheless, the headers with an internal IP address achieve a higher number of access bypasses.

If we narrow our results to hosts that initially responded with a status code only in the `4xx` range rather than any unsuccessful response, we still achieve a successful bypass on over 1,000 hosts. If we narrow down even further to only servers that initially respond with a status code of `403`

TABLE II: Access bypass results. Each column represents the number of 2xx results we receive for each respective source IP address value when we do send a PROXY header. Note that the "unsuccessful" column represents the hosts that initially respond with a status code in the 4xx or 5xx range, or with a connection error.

Header \ No Header	Unsuccessful	4xx	403
2xx Non-internal IP	5,863 (11,640)	648 (3,430)	211 (953)
2xx 127.0.0.1	6,609 (12,235)	1,041 (3,773)	252 (978)
2xx 10.0.0.10	7,480 (13,107)	1,029 (3,762)	246 (974)
2xx 172.16.0.10	7,660 (13,313)	1,011 (3,754)	235 (961)
2xx 192.168.0.10	5,899 (11,527)	1,027 (3,766)	239 (967)

Forbidden, we achieve a successful bypass on over 200 hosts.

Interestingly, these servers are primarily located in China (28.5%), US (16.8%), and South Korea (9.1%). Contrary to the sensitive web pages, these servers are spread across a higher variety of ASs, though mostly operated by Chinese network and telecom providers.

Note that we have to limit our analysis of these access bypass cases to the status codes of the response. As explained in Section IV, we want to avoid leaking sensitive information and therefore used a HEAD request to demonstrate the potential of an access bypass without accessing the corresponding data.

Interestingly, we find that a large portion of our targets return a non-successful response to the malformed probe. Table II shows this number in parentheses for each experiment. This result was later clarified during our responsible disclosure effort (see Section VIII): after interacting with several system administrators, they explained that, although returning a 200 OK status code, these pages display an error message. Their reasoning is that the standard HTTP error codes were not expressive enough for the error handling of their applications. Therefore, we mark them as false positives in our results. Although we are not certain that this is a valid explanation for all results, we still mark each occurrence as a false positive to avoid over-reporting on this problem.

Notably, when we try to achieve an access bypass using the X-Forwarded-For HTTP header field, we only retrieve 25 successful responses. That is, if we spoof the same addresses for this specific HTTP field we are significantly less successful. This highlights the novelty of our attack.

D. Honeypot Experiment

To understand whether these misconfigurations are actively searched for, and abused, by adversaries in the wild, we set up a honeypot infrastructure identical to the simulation environment of Section III. Since both our backend servers require the PROXY header at the start of each connection, any direct access to them without such a header will result in an error. The goal of this setup is to capture whether adversaries attempt to craft a PROXY protocol header, possibly altering the source IP address field. After running our setup in the wild for a period of three weeks, we received a total of 156,122 connections, none of which contained the PROXY header. Any request containing such a header stemmed directly from our load balancer, showing that attackers are unaware of this technique and that this is a novel attack vector.

VI. SMTP Results

We perform a large-scale scan on the SMTP protocol, quasi-identical to our HTTP study, across the IPv4 address range. We first search for open SMTP ports (port 25) using ZMap, and launch the same follow-up probes as for HTTP: one regular probe, five probe with a PROXY header (each containing a different source IP address value), and one probe with a malformed header (see Section IV for more details).

We consider a response successful if the received SMTP status code falls within the 2xx range. This is a conservative lower bound as we want to avoid cases where the PROXY header is seen as an error by the SMTP server, but the server does not abort the connection. From the 9,915,564 hosts with an open port 25, we find 2,332,377 hosts that respond successfully when presented with a PROXY header (even though we are not a trusted proxy server). In total, 25,366 hosts do not accept our connection request without the PROXY header, yet respond successfully when we add a PROXY header.

Many SMTP servers respond with their domain name upon an initial connection. Analyzing these domain names, we encounter 268 instances that are hosted on a .home TLD. This TLD has been deprecated by ICANN in 2018 due to its ambiguity with internal domain names (i.e., many organizations using the TLD for their internal network) [18]. As a result, no public domain today can be hosted on the .home TLD. Therefore, these 268 instances must be within an internal network. This provides additional proof that we can use the PROXY protocol to bypass reverse proxies and reach internal infrastructures.

A. Open Relays

Open relays are mail servers that forward any received email, regardless of the origin, and without verification. While this used to be the default mail server behavior on the Internet, the community soon realized that this mechanism can be abused by spammers and criminals: using an open relay, an adversary can impersonate any email address on the Internet, making it dangerously easy to fool their victims.

Nowadays, open relays rarely occur on the Internet thanks to the many efforts of spam detectors. Furthermore, it is no longer the default setting in mailing software to configure the server as an open relay. Rather, a server will have a specific IP range from which it will accept email requests. In Postfix, for example, the default for this is the localhost address (127.0.0.1).

If an administrator configures their SMTP server behind a proxy server using the PROXY protocol, they have two options for checking the origin IP address: using the packet source IP address, or the source IP address provided in the PROXY header. In case of the latter, we can spoof the header value, and fool the origin check of the server. To determine whether this happens in the wild, we take all hosts that responded with an error during our scan without the PROXY header, but returned a successful response when probed using the PROXY header,

spoofed with `127.0.0.1` as the source IP address (16,779 hosts in total). For these hosts, we attempt to send an email to ourselves using again the spoofed source IP address value of `127.0.0.1` in the PROXY header. We send the email using our private email as the "From" address (which is under the Microsoft Outlook environment), and our university email as the destination address (which is hosted by Google Gmail).

In total, we found **373 SMTP servers that successfully relayed our email without any further access checks or verification steps.** Because of the spoofed value in the PROXY header, the server treats the SMTP packet as if it originated from `127.0.0.1` and, hence, does not proceed with verification checks on the sender address used in the email. Note that for the malformed probes, we receive again zero successful responses, showing that the open relays are indeed due to PROXY header injection and spoofing.

We notice again that the majority of these servers reside in the US (51.47%), this time followed by France (7.5%). They are spread across 49 different autonomous systems, all primarily operated by hosting companies such as Amazon, UDomain, and Hetzner, among others.

Open Relay Scanners and SMTP Safeguards: Unfortunately, open relay scanners do not test for a bypass using the PROXY header, leaving all of these servers undetected by current measures. Normally, open relay detectors actively scan and detect email servers that enable the delivery of spoofed emails. In fact, when we tested this methodology on our own server, our open relay without PROXY protocol configuration was detected within minutes, and immediately put on a spam list, demonstrating how quickly open relays are usually detected. The PROXY protocol circumvents this. Thus, an adversary can impersonate any email address using these mail servers, creating a significant security risk. In Section VIII, we describe our responsible disclosure to the involved parties.

While the Sender Policy Framework (SPF) check failed in our experiments, Gmail still accepted all emails. In fact, this is a common policy among Mail Transfer Agents (MTAs) as many email servers lack, or have incorrect, SPF configuration. Denying every email that fails an SPF check can therefore lead to many false positives. A recent large-scale study has shown that 44 out of 47 providers deliver emails with failed SPF checks [3].

Attack Scenario: With the PROXY protocol, an attacker suddenly obtains access to *persistent* open SMTP relays, which can be used for targetted spoofing attacks and phishing campaigns. We depict in Figure 6 how such an attack might look like. First, the attacker creates a spoofed email that uses as the "From" address the email address of the CEO of a company. Before sending the email to the vulnerable SMTP server, the attacker establishes a connection with the server over SMTP, and uses the PROXY protocol to impersonate a proxy server relaying requests from a client. Moreover, in the PROXY header, the attacker spoofs the source IP address using the `localhost` address (`127.0.0.1`). When the vulnerable server parses the PROXY header, it grabs the source IP address value, and treats the connection as if it were coming from that source IP address provided in the PROXY header. In our attack scenario, this means that the server will treat the connection

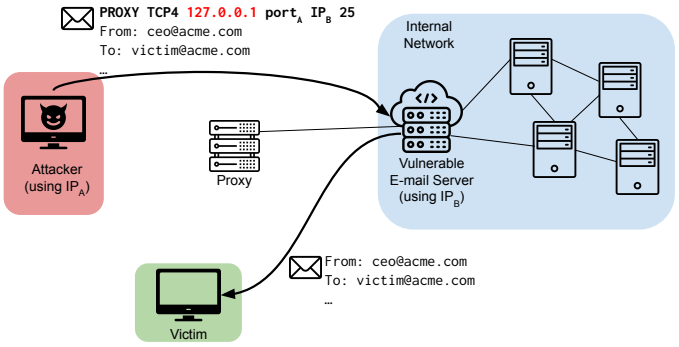


Fig. 6: Example of an attack using PROXY protocol spoofing over SMTP. During the connection setup, the attacker includes a PROXY header where the source IP address value is spoofed to `127.0.0.1`. The vulnerable server will then treat the connection of the attacker as if it was coming from the IP address `127.0.0.1` which is the `localhost` address of the machine. As such, when the attacker sends a command to send an email impersonating the CEO of a company, the vulnerable server will assume it was sent from the server itself, and will, hence, not validate the credentials of the sending email address. As a result, the vulnerable server will forward the email to the victim. The victim will not be able to distinguish whether the email was sent by the attacker or the actual CEO. Note that in this figure the vulnerable server resides within an internal network. This is not a requirement for the attack, as any vulnerable server will be able to forward the email. However, our results show that many servers accepting the PROXY protocol are indeed part of an internal network, as demonstrated by the many `.home` hosts.

from the attacker as if it stems from IP address `127.0.0.1`. Because this is the `localhost` address, the server assumes the email originates from the SMTP server itself and, therefore, does not verify the credentials of the sending email address (`ceo@acme.com` in our example). As a result, the email is sent to its intended destination (the victim), after which the attack is completed. That is, the attacker could successfully impersonate the email address of `ceo@acme.com`, and use it to send a targeted phishing email to its victim. Because these open relays are persistent, this attack is easy to scale, allowing the attacker to launch large-scale campaigns. Our study finds 373 vulnerable servers, demonstrating that an attacker can spread its efforts across multiple servers across the Internet.

VII. SSH Results

We find 21,711,989 hosts listening on port 22, of which 2,343,420 complete a successful handshake when we include a PROXY header. In 30,882 instances, the host responds with an error upon receiving a request not containing the PROXY header, yet returns a successful response when the request does contain the PROXY header. This shows that the PROXY protocol is used beyond HAProxy's known list of supported software.

We compare the banners returned by each server in our target set. Servers accepting the PROXY protocol primarily use Dropbear software (92.0%) whereas servers not responding to the PROXY protocol tend to favor OpenSSH (51.3%, with

Dropbear at 10.3%). We observe no significant difference with respect to the software *versions* used in both groups. That is, both groups have a similar distribution of outdated versions compared to recent versions. This for Dropbear and OpenSSH, as well as any other software encountered. Both groups almost exclusively use SSH version 2, with less than 4% using version 1. We do notice that 20.0% of the servers responding to the PROXY protocol include a weak cipher suite (i.e., using Blowfish, DES, or RC4), making them vulnerable to downgrade attacks. This percentage is lower for regular servers. where we see only 15.0% advertising weak cipher suites.

The 30,882 instances that only respond successfully when presented with a PROXY header could further reveal security issues on the backend server due to, for example, SSH brute force attempts caused by potential rate-limit circumvention on the proxy side. However, we did not pursue any further security assessment since these could quickly lead to shell access on the backend infrastructure, which would be against the ethical considerations of this research.

VIII. Responsible Disclosure

Because our measurement study exposed several security issues of PROXY protocol usage in the wild, we made a dedicated effort to contact the affected parties. For the exposed monitoring platforms, home automation systems, IoT control panels, etc., described in Section V, we performed a `whois` lookup on the respective IP addresses to find a contact for responsible disclosure. For each contact, we sent an email, explaining our experiments and the findings. Furthermore, we asked to be brought in contact with the administrator(s) responsible for the affected server. We followed the same procedure for servers affected by an access bypass (Section V-C) and servers exposing an SMTP open relay (Section VI). In case no direct contact was found we contacted the ISP of the IP address. We also reached out to Apache's security team regarding their default setting of interpreting the source IP address in the PROXY header as the actual IP address, without any accept/blocklist measure in place for trusted proxy servers.

From the `whois` records, we found 301 unique email addresses to contact. 16 emails could not be delivered due to the email address being invalid. From the successfully delivered emails, we received back 24 responses: 7 regarding the exposed platforms, 6 for the access bypass, and 11 regarding the open SMTP relays. Among the affected parties were: a university, a security company, some hosting companies, a major Australian ISP, and some government institutions. This shows that this misconfiguration issue can occur within prominent organizations.

All organizations confirmed the vulnerability. We helped all of them with remediating the issue, and received a bounty offer from 4 of them.

IX. Discussion

The security implications shown in this paper are not a direct cause of the PROXY protocol's design. Rather, they root in implementation details from software vendors, and/or misconfigurations of the system administrator. We discuss further implications of these security issues and how they can be mitigated.

A. False Sense of Security

A crucial difference between the PROXY header and the HTTP `X-Forwarded-For` header lies in how their absence is handled rather than their presence. When a backend server expects a PROXY header, but receives a request without it, the backend server will discard the request, and return an error. For the `X-Forwarded-For` header, servers will generally still accept the request in case the header is not present. As such, the system administrator cannot just hide sensitive content behind a proxy server by relying on the `X-Forwarded-For` header, since the backend server is still reachable. When an administrator enables the PROXY protocol, however, they might assume their content is exclusively accessible through the proxy server, since a direct request to the backend server will result in an error. Our experiments show that this is not the case and that proxy servers can be bypassed by injecting a PROXY header into the request. The PROXY protocol can therefore provide a false sense of security, letting administrators operate under the assumption that their backend servers are not reachable without going through the proxy.

B. Mitigation and Lack of Standardization

Openly accepting PROXY protocol headers from any client is the core of the security issues uncovered in this paper. HAProxy advises users of the PROXY protocol to maintain a list of trusted proxy servers and only accept a PROXY header if it is sent by a proxy server on that list. However, we have seen in Section III that neither NGINX nor Apache has any configuration option in place to maintain such a list. It is therefore up to the system administrator to manually implement and configure such a list (e.g., at the firewall level) to avoid accepting PROXY headers from arbitrary clients. If this is not implemented properly, or simply omitted, any client's PROXY header will be accepted by the backend server. Having a default way of implementing such a proxy server list would fully mitigate any of the security concerns discovered by our research.

In general, we argue that the PROXY protocol in itself lacks a clear standardization. We believe this can be solved by releasing an official Request for Comment (RFC) for the PROXY protocol. Doing so would not only increase the awareness around the protocol but also increase community involvement, leading to better support and standardization.

X. Related Work

To the best of our knowledge, we are the first to study the HAProxy PROXY protocol. Nonetheless, our study relates directly to previous attacks on proxy infrastructures, specifically those involving HTTP header manipulation. Furthermore, we draw parallels between our work and research done on misconfigurations.

A. Proxy Server Security and HTTP Header Manipulation

Mirheidari et al. [23], [24] have shown that Content Delivery Network (CDN) proxies can be manipulated to cache the personal information of a user, which can then later be retrieved from the CDN cache by the attacker. This attack is known as Web Cache Deception (WCD), and the authors demonstrate through a large-scale measurement study how

feasible the attack is in the wild, providing several cases where an adversary could have stolen security tokens.

Another caching attack is the Cache-Poisoned Denial of Service (CPDoS) by Nguyen et al. [27]. In CPDoS, the attacker makes use of HTTP headers to trick the caching proxy into storing error pages from the origin server instead of the actual content of the website. This causes a client to receive an error (such as a `400 Bad Request`) instead of the requested web page that does exist on the website's backend server.

Manipulating HTTP headers goes beyond the realm of CDN caching servers [34]. Mendoza et al. studied header inconsistencies between a website's desktop version versus its mobile variant [22]. In their large-scale measurement study concerning 70,000 websites, the authors identify the security and privacy implications of these header inconsistencies, demonstrating how an attacker could exploit them in the wild.

A common attack vector that uses HTTP header manipulations is request smuggling [15]. With HTTP request smuggling, an attacker crafts a specific HTTP request that will be interpreted differently by the proxy server and the backend server. Concretely, the goal of the attacker is to send multiple requests that will be interpreted as one request by the proxy. The additional (i.e., "smuggled") requests are hence not checked by the proxy, allowing the attacker to bypass any access control checks on the proxy side. This happens by altering the `Content-Length` and/or `Transfer-Encoding` HTTP header fields, which can be interpreted differently depending on which proxy software and backend software is in place. Jabiyev et al. [20] developed a pipeline to test several server combinations for discrepancies in header interpretation that could lead to HTTP request smuggling. They use differential fuzzing and report on a measurement study revealing 23 vulnerable websites in the wild.

Another HTTP header manipulation technique is the so-called "Host of Troubles" attack by Chen et al. [5]. In this attack, the `Host` header field is specifically crafted such that the proxy and the backend server treat the value differently. This can lead to cache poisoning or filter bypassing. Finally, Guo et al. demonstrate how CDN edge servers can be coordinated to launch a targeted DDoS attack [16].

B. Misconfiguration Research

The security issues described in this paper are not a result of the PROXY protocol's properties, but rather of misconfigured infrastructures. The impact of misconfigurations on security has been studied before, and we notice some parallels with the security issues we expose in this paper. Dietrich et al. surveyed several system administrators on the process and origin of security misconfigurations, and how to handle them [12]. Also at scale, we see vulnerabilities measured that originate from a misconfiguration. Continella et al. [10] discovered several S3 cloud storage services from Amazon being unintentionally exposed to the public, and in DNS, misconfigured cycles could lead to server amplification attacks [26]. Pletinckx et al. designed a methodology to detect "orphaned web pages" at scale, which are misconfigured web pages that are no longer linked to by the website hosting them [29]. They show that orphaned web pages tend to be more vulnerable to common attacks such as Cross-Site Scripting (XSS) and SQL Injection. Related to access control, the Baaz tool has been successful at finding access control misconfigurations in file servers [11]. The tool statically analyzes access control lists and identifies inconsistencies among user permissions. Recently, Rahman et al. [30] performed a large-scale analysis of 2,039 Kubernetes manifests to detect security-related misconfigurations. They found over 1,000 misconfigurations, split over 11 security categories. Similarly, Spahn et al. [31] measured exposed container orchestration tools on the Internet, and performed a longitudinal honeypot experiment to investigate attacker trends on Docker, Kubernetes, and common workflow tools.

XI. Conclusion

In this paper, we performed the first Internet-scale measurement study of the PROXY protocol and its security implications. Although backend servers should only accept PROXY protocol information from trused proxy servers, we found more than 170,000 hosts that accept PROXY protocol data from an arbitrary client over HTTP (2,332,377 hosts over SMTP and 2,343,420 hosts over SSH). In over 500 instances, this exposed sensitive web pages, and provided access to IoT monitoring services or smart home control panels, enabling us to regulate household devices without access credentials. Furthermore, we demonstrated how injecting a PROXY header can bypass on-path proxies (and their security mechanisms), allowing a client to directly communicate with the backend server. We found more than 10,000 instances over HTTP in which we can abuse this technique to bypass access control checks by spoofing the source IP address in the PROXY header.

The issues explained in this paper go beyond HTTP, making this a multi-protocol problem. Using PROXY header injection, we bypassed 25,366 proxy servers over SMTP and 30,882 proxy servers over SSH. Moreover, by spoofing PROXY header values over SMTP, we turned 373 email servers into open relays, allowing us to send arbitrary emails from any email address. These servers are currently undetectable by open relay scanners as these scanners do not use the PROXY protocol during their probes, making these open relay servers persistent on the Internet. Finally, we detailed our responsible disclosure procedures and gave recommendations on how to mitigate the identified security misconfigurations.

Acknowledgment

The authors would like to thank Don Kileen for his amazing support during the scans. This material is based upon work supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or its federal agency and industry partners.

References

[1] Akamai, "About application load balancer," https://techdocs.akamai.com/cloudlets/docs/what-app-load-balancer, 2021.

[2] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan, "The menlo report," *IEEE Security & Privacy*, vol. 10, 2012.

[3] B. Blechschmidt and B. Stock, "Extended hell(o): A comprehensive large-scale study on email confidentiality and integrity mechanisms in the wild," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023.

[4] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002. [Online]. Available: https://doi.org/10.1145/509907.509965

[5] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: Multiple host ambiguities in http implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016.

[6] Cloudflare, "mmproxy," https://github.com/cloudflare/mmproxy, 2022.

[7] ——, "The cloudflare global network," https://www.cloudflare.com/network/, 2023.

[8] ——, "How cloudflare works," https://developers.cloudflare.com/fundamentals/concepts/how-cloudflare-works/, 2023.

[9] ——, "What is a reverse proxy? — proxy servers explained," https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/, 2023.

[10] A. Continella, M. Polino, M. Pogliani, and S. Zanero, "There's a hole in that bucket! a large-scale analysis of misconfigured s3 buckets," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018.

[11] T. Das, R. Bhagwan, and P. Naldurg, "Baaz: A system for detecting access control misconfigurations," in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 2010.

[12] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig, "Investigating system operators' perspective on security misconfigurations," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018.

[13] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. USENIX Association, 2013.

[14] T. Fiebig, "Crisis, ethics, reliability & a measurement.network: Reflections on active network measurements in academia," in *Proceedings of the Applied Networking Research Workshop, ANRW 2023, San Francisco, CA, USA, 24 July 2023*. ACM, 2023.

[15] M. Grenfeldt, A. Olofsson, V. Engström, and R. Lagerström, "Attacking websites using HTTP request smuggling: Empirical testing of servers and proxies," in *25th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2021, Gold Coast, Australia, October 25-29, 2021*. IEEE, 2021.

[16] R. Guo, J. Chen, Y. Wang, K. Mu, B. Liu, X. Li, C. Zhang, H. Duan, and J. Wu, "Temporal CDN-Convex lens: A CDN-Assisted practical pulsing DDoS attack," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023.

[17] M. Henzinger, "Finding near-duplicate web pages: A large-scale evaluation of algorithms," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '06. New York, NY, USA: Association for Computing Machinery, 2006. [Online]. Available: https://doi.org/10.1145/1148170.1148222

[18] ICANN, "Approved board resolutions — regular meeting of the icann board 4 february 2018," https://www.icann.org/en/board-activities-and-meetings/materials/approved-board-resolutions-regular-meeting-of-the-icann-board-04-02-2018-en#2.c, 2018.

[19] L. Izhikevich, R. Teixeira, and Z. Durumeric, "LZR: Identifying unexpected internet services," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[20] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-reqs: Http request smuggling with differential fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021.

[21] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: Association for Computing Machinery, 2007. [Online]. Available: https://doi.org/10.1145/1242572.1242592

[22] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering http header inconsistencies and the impact on desktop/mobile websites," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018.

[23] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, "Cached and confused: Web cache deception in the wild," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020.

[24] S. A. Mirheidari, M. Golinelli, K. Onarlioglu, E. Kirda, and B. Crispo, "Web cache deception escalates!" in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 2022.

[25] R. Moskowitz, D. Karrenberg, Y. Rekhter, E. Lear, and G. J. de Groot, "Address allocation for private internets," https://datatracker.ietf.org/doc/html/rfc1918, 1996.

[26] G. C. M. Moura, S. Castro, J. Heidemann, and W. Hardaker, "Tsuname: Exploiting misconfiguration and vulnerability to ddos dns," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[27] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your cache has fallen: Cache-poisoned denial-of-service attack," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2019.

[28] OWASP, "Log injection," https://owasp.org/www-community/attacks/Log_Injection, 2023.

[29] S. Pletinckx, K. Borgolte, and T. Fiebig, "Out of sight, out of mind: Detecting orphaned web pages at internet-scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3460120.3485367

[30] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, may 2023.

[31] N. Spahn, N. Hanke, T. Holz, C. Kruegel, and G. Vigna, "Container Orchestration Honeypot: Observing Attacks in the Wild," in *26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 23)*, October 2023.

[32] H. Technologies, "The proxy protocol," https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt, 2020.

[33] ——, "Use the proxy protocol to preserve a client's ip address," https://www.haproxy.com/blog/use-the-proxy-protocol-to-preserve-a-clients-ip-address, 2022.

[34] G. Tyson, S. Huang, F. Cuadrado, I. Castro, V. C. Perta, A. Sathiaseelan, and S. Uhlig, "Exploring http header manipulation in-the-wild," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017.

[35] F. Valeur, G. Vigna, C. Kruegel, and E. Kirda, "An anomaly-driven reverse proxy for web applications," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06. New York, NY, USA: Association for Computing Machinery, 2006.

APPENDIX

We configure 3 Ubuntu Server 22.04.03 Virtual Machines (VMs):

- A backend server running NGINX 1.18.0, hosting a web page. Static IP address: `192.168.56.21`
- A backend server running Apache 2.4.52, hosting a web page. Static IP address: `192.168.56.22`
- A reverse proxy server running HAProxy 2.4.22, functioning as a load balancer for the two backend servers. Static IP address: `192.168.56.10`

```
frontend http-in
    bind *:80
    default_backend servers

backend servers
    balance roundrobin
    server s1 192.168.56.21:80 send-proxy
    server s2 192.168.56.22:80 send-proxy
```

Listing 3: HAProxy configuration example for a load balancing setup. By setting the `send-proxy` argument, we make sure that the load balancer adds a PROXY header during the connection setup with the backend server.

```
server{
    listen 80 proxy_protocol;
    # Other configurations...
}
```

Listing 4: NGINX web server configuration example. By setting the `proxy_protocol` argument, the server expects a PROXY header to be present during each connection setup with the web server.

```
<VirtualHost *:80>
    # Some configurations...
    RemoteIPProxyProtocol On
    # Other configurations...
</VirtualHost>
```

Listing 5: Apache web server configuration example. By setting `RemoteIPProxyProtocol` to `ON`, the server expects a PROXY header to be present during each connection setup with the web server.