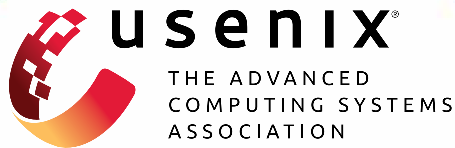


# ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication

Marcus Brinkmann, *Ruhr University Bochum;* Christian Dresen, *Münster University of Applied Sciences;* Robert Merget, *Ruhr University Bochum;* Damian Poddebniak, *Münster University of Applied Sciences;* Jens Müller, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum;* Sebastian Schinzel, *Münster University of Applied Sciences*

https://www.usenix.org/conference/usenixsecurity21/presentation/brinkmann

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

# ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication

Marcus Brinkmann[1], Christian Dresen[2], Robert Merget[1], Damian Poddebniak[2], Jens Müller[1], Juraj Somorovsky[3], Jörg Schwenk[1], and Sebastian Schinzel[2]

[1]Ruhr University Bochum
[2]Münster University of Applied Sciences
[3]Paderborn University

## Abstract

TLS is widely used to add confidentiality, authenticity and integrity to application layer protocols such as HTTP, SMTP, IMAP, POP3, and FTP. However, TLS does not bind a TCP connection to the intended application layer protocol. This allows a man-in-the-middle attacker to redirect TLS traffic to a different TLS service endpoint on another IP address and/or port. For example, if subdomains share a wildcard certificate, an attacker can redirect traffic from one subdomain to another, resulting in a valid TLS session. This breaks the authentication of TLS and *cross-protocol attacks* may be possible where the behavior of one service may compromise the security of the other at the application layer.

In this paper, we investigate cross-protocol attacks on TLS in general and conduct a systematic case study on web servers, redirecting HTTPS requests from a victim's web browser to SMTP, IMAP, POP3, and FTP servers. We show that in realistic scenarios, the attacker can extract session cookies and other private user data or execute arbitrary JavaScript in the context of the vulnerable web server, therefore bypassing TLS and web application security.

We evaluate the real-world attack surface of web browsers and widely-deployed email and FTP servers in lab experiments and with internet-wide scans. We find that 1.4M web servers are generally vulnerable to cross-protocol attacks, i.e., TLS application data confusion is possible. Of these, 114k web servers can be attacked using an exploitable application server. Finally, we discuss the effectiveness of TLS extensions such as Application Layer Protocol Negotiation (ALPN) and Server Name Indiciation (SNI) in mitigating these and other cross-protocol attacks.

## 1 Introduction

**TLS.** With Transport Layer Security (TLS) [56], confidential and authenticated channels are established between two communication endpoints. In typical end-user protocols, such as HTTP, SMTP, or IMAP, the TLS server authenticates to the client by presenting an X.509 certificate. In this setting, the server is identified by the *Common Name* (CN) field or the *Subject Alternate Name* (SAN) extension in the certificate, which contains one or more hostnames or wildcard patterns (e.g., `*.bank.com`). As part of the certificate validation, the client confirms that the destination of the request matches the CN or SAN of the certificate.

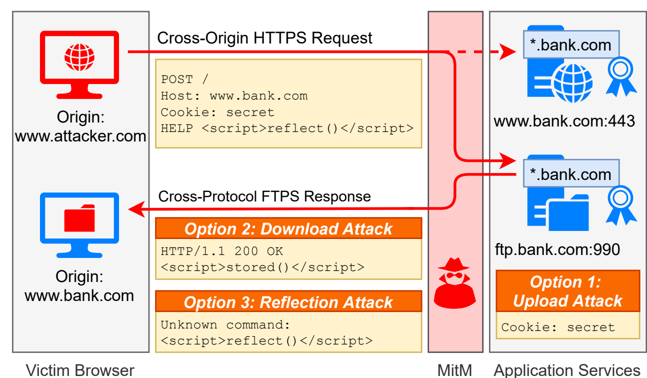


Figure 1: Basic idea behind application layer cross-protocol attacks on HTTPS. A MitM attacker leads the victim to an attacker-controlled website that triggers a cross-origin HTTPS request with a specially crafted FTP payload. The attacker then redirects the request to an FTP server that has a certificate compatible with the web server. The attack either (1) uploads a secret cookie to FTP, or (2) downloads a stored malicious JavaScript file from FTP, or (3) reflects malicious JavaScript contained in the request. In case (2) and (3), the JavaScript code is executed in the context of the targeted web service.

Since TLS does not protect the integrity of the TCP connection itself (i.e., source IP & port, destination IP & port), a man-in-the-middle (MitM) attacker can redirect TLS traffic for the *intended* TLS service endpoint and protocol to another, *substitute* TLS service endpoint and protocol. If the client considers the certificate of the substitute server to be valid for the intended server, for example, if wildcard certificates

are shared among subdomains, the authentication of the connection is violated. This can enable *cross-protocol attacks* at the application layer, where the client unknowingly sends the protocol data for the intended server to the substitute server that expects a different protocol, potentially compromising the security of either server at the application layer.

In general, cross-protocol attacks can be considered between any two TLS-secured protocols. Although some protocol combinations will more likely lead to successful attacks than others, even wildly different data formats can be interoperable. For instance, a length-specified binary protocol can be embedded in HTTP [53]. The number of potential attack scenarios exhibits quadratic growth with the number of application protocols and implementations.

**Cross-Protocol Attacks on HTTPS.** Cross-protocol attacks on unprotected HTTP were first described by Jochen Topf [59], and we refer to Section 10 for the history of these plaintext attacks. The first cross-protocol attack on a web server secured by TLS was found by Jann Horn (with input from Michał Zalewski) [38], who demonstrates reflected and stored Cross-Site-Scripting (XSS) attacks using an exploitable FTP server. Horn considered a MitM attacker who also has man-in-the-browser (MitB) privileges by serving a web page with malicious JavaScript to the victim, as shown in Figure 1. This is the classical attacker model for attacks on the TLS Record Layer, such as BEAST [19], CRIME [20], POODLE [45], and Lucky 13 [4]. Using the MitB, the attacker triggers an HTTPS POST request to the target web server, where the body contains valid FTP commands. Using the MitM, the attacker redirects the request to an FTP server with a compatible certificate. The browser completes the TLS handshake with the FTP server and sends the HTTP request as application data. If the FTP server is error-tolerant, it may ignore invalid data such as the HTTP header and execute the valid FTP commands in the HTTP body. In this example, the payload is `HELP <script>reflect()</script>`. An exploitable FTP server reflects the embedded JavaScript to the client in an error message. Although the response from the FTP server is not valid HTTP, an exploitable browser finds the JavaScript by content sniffing [8] and executes it within the context of the original request, completing the attack [38].

We note that during this attack, each component (browser, web server, and application server) functions as expected and the security violation is an emergent property due to the attacker's ability to recombine the components in an unintended way. The root cause is the inability of the TLS authentication mechanism to prevent the confusion in the first place.

**Systematic Analysis.** In this work, we investigate cross-protocol attacks on TLS in general and on HTTPS in particular with a *case study*, where we target a web server for which the user may have established some kind of privileged session (i.e., the user is logged into an account). For example, this could be a webmail server (e.g., Roundcube) colocated with an email server or a content management system (e.g., WordPress) colocated with an FTP server. Systematically extending the above attack example from [38], we consider cross-protocol attacks on the web server that redirect cross-origin HTTP requests to SMTP, IMAP, POP3, or FTP servers, using one of three attack methods: 1. In an *upload* attack, the attacker tricks the victim into uploading secret session data contained in the request (i.e., a cookie). 2. In a *download* attack, the attacker prepares a stored XSS payload at the application server and tricks the victim into downloading it. 3. In a *reflection* attack, the attacker returns a cross-protocol response with a reflected XSS payload included in the request. In the two XSS methods, the script executes in the context of the request and can be used to extract secrets or launch malicious same-origin requests with the authority of the user, breaking the security of the web application.

The protocols SMTP, IMAP, POP3, and FTP were selected because they are line-based text protocols similar to HTTP, widely deployed on the Internet, likely to be configured with certificates that are compatible with public web servers, and mature enough to minimize any potential risks to the infrastructure by our internet-wide scans.

*Attack variation.* Under restrictive conditions, these attacks can even succeed in a pure web attacker model (MitB, but not MitM). For this, the application server must have the same hostname as the web server and the browser must not include the port number in the Same-Origin-Policy (SOP). For the majority of this work, we assume a MitM+MitB attacker and revisit the pure MitB model in Section 8.

**Evaluation.** In practice, cross-protocol attacks are sensitive to many requirements, such as certificate compatibility, ability to upload, download, or reflect data, and application tolerance towards syntax errors caused by mixing two protocols in one channel. In our case study of cross-protocol attacks on HTTPS, using SMTP, IMAP, POP3, and FTP application servers, we address these concerns in three evaluations.

1. We identified 25 popular SMTP, IMAP, POP3, and FTP implementations and evaluated their suitability for cross-protocol attacks on HTTPS in a series of lab experiments. We found that 13 are exploitable with at least one attack method (see Table 3). We also implemented a full proof-of-concept that demonstrates all three attack methods on a well-secured web server, using exploitable SMTP, IMAP, POP3, and FTP application servers.

2. We evaluated seven browsers for their error tolerance. We find that Internet Explorer and Edge Legacy still perform content sniffing and thus are vulnerable to all presented attacks, while all other browsers allow at least FTP upload and download attacks (see Table 1).

3. In an internet-wide scan, we collected X.509 certificates served by SMTP, IMAP, POP3, and FTP servers. We analyzed how many of these are likely to be trusted by

major web browsers. For each certificate, we extracted the hostnames in the CN field and SAN extension and checked if there exists a web server on these hosts. We found 1.4M web servers that are compatible with at least one trusted application server certificate, making them vulnerable to cross-protocol attacks (see Table 4). Of these, 119k web servers are compatible with an application server that is exploitable in our lab settings.

**Countermeasures.** We present generic countermeasures to all cross-protocol attacks, based on the ALPN and SNI extensions. These countermeasures solve the issue at the TLS layer and can be deployed without backward compatibility issues.

**Contributions.** We make the following contributions:

- The first generic description of cross-protocol attacks against TLS applications.
- A systematic case study of cross-protocol attacks against HTTPS, exploiting popular SMTP, IMAP, POP3, and FTP application servers.
- A complete IPv4 scan for web and application servers allowing such cross-protocol attacks, measuring the number of vulnerable and exploitable services.
- Analysis of generic countermeasures to cross-protocol and related content confusion attacks with minimal changes to implementations and standards.

**Responsible Disclosure and Artifact Availability.** We reported our findings to TLS libraries, exploitable application servers, and our national CERT. We will publish all source code used in the evaluation of this paper as Open Source at: https://github.com/RUB-NDS/alpaca-code.

# 2 Background

## 2.1 TLS and X.509 Certificates

Transport Layer Security (TLS) is a cryptographic layer between the transport layer (i.e., TCP) and an application layer protocol [17]. The TLS protocol consists of two phases. In the first phase, the client and server perform a TLS handshake to exchange used versions, randoms, cryptographic algorithms, and supported extensions, in order to derive symmetric keys. In the second phase, the symmetric keys are used to protect application data, such as HTTP, SMTP, IMAP, POP3, or FTP.

**STARTTLS.** Unprotected legacy protocols can be extended to support TLS by upgrading a plaintext connection using a protocol-specific STARTTLS command. After the TLS handshake succeeds, the legacy protocol is continued in the encrypted application data. STARTTLS was first standardized in RFC 2487 [35] as an extension to SMTP.

**Server Certificates.** In a typical TLS scenario on the web, a server authenticates to a TLS client with an X.509 certificate [14] during the handshake. The certificate contains the server public key (which is used within the handshake), server domain name, expiration date, and several extensions. For example, there exist extensions for defining key usage (e.g., signing and encipherment), extended key usage (e.g., WWW server protection), and locating the certificate revocation list. For the security of the connection, it is crucial that the TLS client validates all these certificate properties.

**Server Name Indication (SNI).** In some deployments, several web (or other) services may be hosted at the same IP address and port. To support this *virtual hosting* configuration, the client indicates the desired hostname in the *Server Name Indication* (SNI) extension [22]. While SNI is well-supported by HTTPS servers, it is much less common in other application protocols such as SMTP, IMAP, POP3, and FTP.

**Application-Layer Protocol Negotiation (ALPN).** The ALPN extension [27] allows the TLS peers to select a specific application layer protocol. With ALPN, a web server can offer different application layer protocols on the same port, for example, more performant versions of the HTTP protocol (in particular, HTTP/2 along with HTTP/1.1), while avoiding additional round trips for protocol negotiation. The client sends the protocols it supports as a list of strings to the server, and the server selects a protocol it supports or sends an alert if no protocol supported by the server is found among the list. Protocol names are presented in ASCII and assigned by IANA.[1] The names for HTTP, IMAP, POP3, and FTP have already been standardized, while SMTP is not yet registered.

## 2.2 Application Layer Protocols

**HTTP.** The *Hypertext Transfer Protocol (HTTP)* [24] is a line-based text protocol for the World Wide Web, which is typically accessed with a browser. Web servers are secured by TLS and other safeguards to protect sensitive user data. For example, cookie attributes (e.g., `Secure`, `HttpOnly`, or `SameSite`) protect authentication tokens, Content Security Policy (CSP) protects against cross-site scripting attacks, and the Same Origin Policy (SOP) mitigates cross-domain interaction risks, while Cross-Origin Resource Sharing (CORS) enables sharing across domains in a controlled manner.

**Email Protocols and FTP.** The *Simple Mail Transfer Protocol (SMTP)* [40] is used to send emails. *Post Office Protocol (POP3)* [47] and *Internet Message Access Protocol (IMAP)* [15] are used to access them. The *File Transfer Protocol (FTP)* is a protocol to upload and download arbitrary files to a server [52]. All protocols are line-based text protocols. TLS can be used either explicitly by upgrading an insecure

[1] https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

connection using the STARTTLS (or similar) command or implicitly by connecting on an alternative port. A peculiarity of the SMTP protocol is that after sending a command, the client should wait for the server's response before sending the next one unless the server supports *command pipelining* [26].

Note that FTP uses separate connections for control and data transfer. The client first opens a control connection, then sends a command to open a data port in the server. Then it establishes an implicit TLS connection to that port to upload or download files (aka *passive mode*). We assume the FTP server enforces TLS session resumption on the data connection to protect against well-known port stealing attacks [5] by binding the data channel cryptographically to the control channel, as first proposed and implemented by Chris Evans [23].

## 3 TLS-Based Cross-Protocol Attacks

In a generic cross-protocol attack, we assume a client $C$ and two application servers $S_{int}$ and $S_{sub}$. The client $C$ uses protocol $A$ with the *intended* server $S_{int}$. The *substitute* server $S_{sub}$ uses an unrelated protocol $B$ and runs on a different TCP endpoint (IP, port). However, $S_{sub}$ has a certificate that is *compatible* with $S_{int}$, i.e., the certificate could be used by $S_{int}$ in place of its regular certificate without breaking the intended connection from the client. The goal of the attacker is to trick either $S_{sub}$ into accepting application data from $C$ or to trick $C$ into accepting application data from $S_{sub}$. Because $C$ and $S_{sub}$ use different protocols, this type of attack is called a *cross-protocol* attack. The attack works like this:

1. The man-in-the-middle (MitM) attacker interposes the TCP connection between $C$ and $S_{int}$, and forwards all data from $C$ to $S_{sub}$ and vice versa. Optionally, the MitM first creates a new TLS endpoint on $S_{sub}$ with STARTTLS and only then forwards the TLS traffic from $C$.

2. The application server $S_{sub}$ performs the TLS handshake with the client $C$ and presents its certificate $\text{Cert}_{sub}$.

3. Because $\text{Cert}_{sub}$ is compatible with $S_{int}$, the client $C$ accepts it and completes the TLS handshake. Subsequently, $C$ will send application data to $S_{sub}$.

4. $S_{sub}$ tries to interpret the data sent by $C$. Because the client sends application data in format $A$ and the $S_{sub}$ expects format $B$, this may result in security violations.

5. If $S_{sub}$ responds by sending, e.g., error messages to $C$ in format $B$, the client processes these as if they were in format $A$, which may also result in security violations.

The presented attack breaks the authentication of the connection in Step 3, when $C$ finishes the handshake with $S_{sub}$, as $C$ is not noticing that it performed the handshake with $S_{sub}$ instead of $S_{int}$. In this case we say that $S_{int}$ and $S_{sub}$ are *vulnerable to cross-protocol attacks*. This loss of authentication can lead to severe security issues at the application layer and potentially a loss in confidentiality, for example if $S_{sub}$ writes application data to a log file readable by the attacker.

In practice, cross-protocol attacks are hindered by a series of obstacles. We identify the following requirements for cross-protocol attacks to be exploitable:

- *TLS Compatibility.* The client $C$ and application server $S_{sub}$ must complete the TLS handshake, and $C$ must accept the certificate of the substitute server as valid for the intended server. We provide more details on this requirement in Subsection 3.1.

- *Tolerance To Protocol Embedding.* $S_{sub}$ should tolerate a certain amount of invalid traffic that comes from protocol $A$ in which the payload for $S_{sub}$ in format $B$ is embedded. Likewise, if the client is expected to process any response by $S_{sub}$, it should tolerate a certain amount of invalid traffic that comes from protocol $B$ in which the payload for $C$ in format $A$ is embedded.

- *Application Server Exploitability.* $S_{sub}$ must provide some feature, mechanism, or behavior supporting the attack. The details are protocol- and implementation-specific, but generally, the exploited behavior will be unexpected by $C$ and differ considerably from the behavior of $S_{int}$, resulting in some form of security violation.

### 3.1 TLS Compatibility

To enable cross-protocol attacks, the server $S_{sub}$ must provide a certificate and TLS configuration that is compatible with the certificate and configuration of $S_{int}$ such that $C$ will successfully complete a handshake with $S_{sub}$ in a MitM scenario. We will now describe the most important requirements for this.

**Certificate Names.** The client application must accept the server names in the certificate of $S_{sub}$ as valid for $S_{int}$. This can be the case for one of two reasons: 1. The certificate presented by $S_{sub}$ has the hostname of $S_{int}$ in the CN field or SAN extension, or 2. The certificate of $S_{sub}$ is a wildcard certificate that matches the hostname of $S_{int}$ (e.g., `*.bank.com` matches `www.bank.com`). Such configurations occur spontaneously when an administrator, unaware of the risk of cross-protocol attacks, deploys a multi-domain or wildcard certificate to save costs and administrative effort, or simply copies a web server certificate to another application server to support opportunistic encryption without validation, as is common for non-HTTPS services [36]. Note that for the success of cross-protocol attacks it is not required that the certificate presented by $S_{sub}$ is valid for $S_{sub}$ itself.

**Certificate Validity.** The certificate must also satisfy a broad range of other conditions to be considered valid by $C$. Most importantly, the certificate must be signed by a certificate authority trusted by $C$, and the certificate must not be expired.

Note that there is no possibility to define the designated application layer protocol in an X.509 certificate. While a certificate can include an extended key usage extension, this extension can only indicate a very broad purpose for which the certified public key may be used [14], for example, code signing, client/server authentication, or email protection.

**TLS Handshake Parameters.** The TLS protocol itself can also cause the attack to fail. In particular, the client $C$ must support at least one TLS version and cipher suite offered by $S_{sub}$ which may differ from those provided by $S_{int}$. This is especially important if the two protocols adopt new versions, cipher suites, and extensions at different speeds, for example if the client $C$ deprecates some features before $S_{sub}$ is updated to support suitable replacements.

**ALPN.** If the client $C$ supports the ALPN extension, it will send only the ALPN identifiers for the intended protocols. If $S_{sub}$ is also implementing the ALPN extension, it cannot choose a matching application layer protocol. In this case, the ALPN extension mandates to abort the handshake and send a fatal TLS alert message [27]. Thus, at first glance, this might prevent the attack. However, ALPN was never considered as a security feature, but merely as a mechanism to multiplex different protocols on the same TCP endpoint [27]. If the substitute server is unaware of ALPN, or the ALPN extension is not transmitted by $C$, or a failure in ALPN negotiation is silently ignored by the server, the handshake proceeds despite ALPN. We will show in Subsection 7.2 that this is often the case for servers implementing protocols other than HTTPS.

**SNI.** A client may specify the server name in the SNI extension. If $S_{sub}$ is not responsible for resources on the indicated server name, it can reject the connection and thus prevent the attack. However, cross-protocol attacks are not affected by SNI if two services run under the same name, or if the substituted server does not implement it, or if the server is misconfigured. Similar to ALPN, SNI was not specified as a security feature, but to multiplex different virtual servers (possibly implementing the same protocol) on the same TCP endpoint. It is still not widely supported outside of HTTP. We will show in Subsection 7.2 that servers implementing protocols other than HTTP often ignore the SNI extension.

## 4 Cross-Protocol Attacks on HTTPS

We now consider only cross-protocol attacks where the client $C$ is a web browser using HTTPS. The goal of the attacker is either to execute JavaScript in the context of the targeted web server, leading to Cross-Site-Scripting (XSS), or to steal the session cookie of an already logged in user. After describing our attacker model, we will discuss specific requirements for such cross-protocol attacks against HTTPS.

As is common for TLS attacks [4, 19, 20, 45], we consider a more powerful MitM attacker, who is also a man-in-the-browser (MitB) with the ability to execute arbitrary JavaScript in the context of a website controlled by the attacker. The MitB attacker can force the client to send cross-origin requests to the targeted web server but is unable to read the response due to the Same-Origin-Policy (SOP). However, the attacker can control some parts of the HTTP header, and, in the case of the `POST` method, the complete body of the request. In a simple cross-protocol scenario, the attacker accepts the headers as given and places a malicious payload in the body. If the substitute server tolerates the HTTP header, it will eventually reach the body of the request and process the malicious payload, giving the attacker full access to the features provided by the substitute server protocol and implementation.

### 4.1 Attack Methods

Through an extensive review of the existing documentation of browser-related cross-protocol attacks, we identified three general methods the attacker can use within application layer protocols to attack HTTPS sessions.

**Upload Attack.** For this attack, we assume the attacker has some ability to upload data to $S_{sub}$ and retrieve it later. In an upload attack, the attacker tries to store parts of the HTTP request of the browser (specifically the `Cookie` header) on $S_{sub}$. This might, for example, occur if the server interprets the request as a file upload or if the server is logging incoming requests verbosely. After a successful attack, the attacker retrieves the content on the server independently of the connection from $C$ and retrieves the HTTPS session cookie.

**Download Attack – Stored XSS.** For this attack, we assume the attacker has some ability to store data on $S_{sub}$ and download it. In a download attack, the attacker exploits benign protocol features to download previously stored (and specifically crafted) data from $S_{sub}$ to $C$. This is similar to a stored XSS vulnerability. However, because a protocol different from HTTP is used, even sophisticated defense mechanisms against XSS, like the Content-Security-Policy (CSP) [64], can be circumvented. Very likely, $S_{sub}$ will not send any CSP by itself, and large parts of the response are under the control of the attacker.

**Reflection Attack – Reflected XSS.** In a reflection attack, the attacker tries to trick the server $S_{sub}$ into reflecting parts of $C$'s request in its response to $C$. If successful, the attacker sends malicious JavaScript within the request that gets reflected by $S_{sub}$. The client will then parse the answer from the server, which in turn can lead to the execution of JavaScript in the context of the targeted web server.

### 4.2 Web Browser Tolerance

With reflection and download attacks, the data returned by $S_{sub}$ will often not be a proper HTTP response but contain 'noise' in the form of a banner identifying the application

server, as well as any syntax errors and other messages output by the server processing the HTTP request. This is particularly significant for the very beginning of the response, where the web browser expects an HTTP response status line. If this line is missing, a browser may assume HTTP/0.9, which allows a *Simple-Response* [10, ch. 4.1] that does not contain any headers. Without headers specifying the content type, the embedded JavaScript will only be executed if the the browser interprets the response as HTML due to *content-sniffing* [8].

### 4.3 Application Server Error Tolerance

Sending an HTTP request to a non-HTTP server will likely cause syntax errors due to differences in the protocol languages. Thus, it is an advantage for the attacker if $S_{sub}$ is "liberal" in what it accepts in the sense of Postel's Law.[2]

An HTTP request consists of four parts: 1. The HTTP request line with method, URI, and version, 2. zero or more `key: value` header fields, 3. an empty line separating the header from the body, and 4. the attacker-controlled body of the request containing the POST data. For a successful attack, we require that the application server keeps processing commands even after encountering the initial HTTP request line and up to a certain number of HTTP header lines. Some application servers terminate the connection after some number of syntax errors. If this number is too low (i.e., smaller than or equal to the number of lines in the header), our attack will probably fail because the POST data in the request will never be processed. The exact number of header lines that must be processed without terminating the connection depends on the web browser of the victim. For example, Chrome 83 sends 17 header lines as part of a POST request (see Table 1).

Additionally, some application servers (e.g., Postfix SMTP) specifically try to detect cross-protocol attacks by recognizing common HTTP method tokens in the request line.

### 4.4 Advanced Exploitation Techniques

So far, we only considered cross-protocol attacks that contain a single message exchange. However, the order of the content of an HTTP request is fixed: first comes the header, potentially containing a sensitive cookie that the attacker wants to steal, and then comes the body of the request with the malicious payload. The attacker has only a few options (such as the path in the URL and some header lines) to affect the request content before the critical header line with the cookie. This makes some cross-protocol attacks using only a single HTTP request very challenging.

If the attacker is able to force several HTTP requests within a single connection, more sophisticated cross-protocol attacks may become practical. Using multiple requests allows the attacker to send a malicious payload to $S_{sub}$ in the first request to prepare $S_{sub}$ into a state that allows the second request to complete the attack. This strategy is especially useful for upload attacks, where the first request prepares $S_{sub}$ in such a way that the cookie in the second request is uploaded to the server. However, in order to send *two* HTTP requests inside a single TLS connection, the browser has to reuse the connection. We evaluate when this is the case in Section 5.

Even more powerful attacks are possible if $S_{sub}$ is vulnerable to a TLS renegotiation attack [55], which allows an attacker to prepend arbitrary bytes to the victim plaintext data, bypassing all potential protocol countermeasures and intolerances. This way, application data of the client gets interpreted in the attacker session such that the attacker can prepare an arbitrary prefix for transmitted application data.

[2] Also called the *robustness principle*: "be conservative in what you do, be liberal in what you accept from others." [51]

| **Results** | Chrome | Firefox | IE | Edge Legacy | Edge | Opera | Safari |
|---|---|---|---|---|---|---|---|
| **Header Lines** | 17 | 11 | 11 | 12 | 17 | 17 | 11 |
| **Content Sniffing** | ○ | ○ | ● | ● | ○ | ○ | ○ |
| **Keep-Alive** | | | | | | | |
| w/ HTTP/1.1 | ● | ● | ● | ● | ● | ● | ● |
| w/ noise | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

● Support ○ No support

Table 1: Browser behavior relevant to cross-protocol attacks.

## 5 Evaluation of Web Browsers

We evaluated the browser behavior relevant to cross-protocol attacks for Chrome 86, Firefox 81, Internet Explorer 11, Edge Legacy 44, Edge 86, Opera 71, and Safari 14 by manually accessing a custom web server with one test page for each property under evaluation. The results are shown in Table 1.

**Number of Header Lines.** For each browser, we determined how many header lines are included in a typical POST request sent by the browser. This is also the minimum number of errors an application server with a line-based text protocol must tolerate to be usable for cross-protocol attacks. We find that all browsers send headers that consist of 11 to 17 lines.

**Content Sniffing.** As described in Subsection 4.2, reflection and download attacks can be sensitive to noise in the protocol data returned by the application server. For each browser, we evaluated if the browser performs content sniffing and executes embedded JavaScript anyway. We could confirm that this is indeed the case for Internet Explorer and Edge Legacy, while all other tested browsers do not perform content-sniffing and thus do not execute JavaScript in noisy responses.

**Connection Reuse.** As mentioned in Subsection 4.4, sending more than one request in a single cross-protocol connection can be advantageous for an attacker. The HTTP/1.1 standard

defines persistent connections between client and server for multiple HTTP requests by default [24]. Thus, the browser should reuse the TCP connection as long as the HTTP version of the server response is at least 1.1 and the server did not send the HTTP header *Connection: close* [24, Section 8.1.2.1].

We tested this behavior for Chrome, Edge, Edge Legacy, Firefox, Internet Explorer, Opera, and Safari. All browsers reuse the connection after receiving a valid HTTP/1.1 response containing at least the *Status-Line* and a *Content-Length*. Chrome, Firefox, and Opera even accept a *Status-Line* only consisting of the HTTP-Version without a *Status-Code* or a *Reason-Phrase*. Internet Explorer and Edge require a complete *Status-Line*. None of the browsers reuse the connection if the first line of the response does not begin with the token `HTTP`. This is relevant for upload attacks relying on connection reuse that include protocol noise at the beginning of the response, defeating the goal of sending more than one request in the same TLS connection.

## 6 Evaluation of Application Servers

In this section, we demonstrate how cross-protocol attacks on HTTPS can be executed using SMTP, IMAP, POP3, and FTP as application servers. We first describe how upload, download, and reflection attacks can be realized with these protocols. Then, we analyze 25 popular implementations of these protocols and evaluate whether they are exploitable by cross-protocol attacks against HTTPS in at least one browser.

### 6.1 Attack Strategies

We identified the following attack strategies to realize upload, download, and reflection attacks on HTTPS using SMTP, IMAP, POP3, and FTP application servers.

**Reflection Attacks.** All protocols in our case study are line-based protocols. They interpret each line of the HTTPS request as a command and will generate a response for each. If an implementation receives a command, it may use some data from the input in its response. For example, sending `HELP <script>attack();</script>` to an FTP server may lead to the response `Unknown command: <script>attack();</script>` (see Figure 1). Usually, as is the case for SMTP, POP3, and FTP, the availability of such reflection vectors is an implementation artifact, depending on the verbosity of error messages and other factors. But in IMAP, every command must begin with a so-called *tag*, which must be reflected to allow the client to match the server response to the issued command [15, Section 2.2.1]. Although this reflection vector is mandated by the protocol standard, the allowed character set may differ between implementations.

Reflected responses are likely to contain some 'noise' before and after the reflected payload. In this case, the browser has to support content-sniffing to allow a reflected XSS attack.

**FTP Upload and Download Attacks.** FTP uses two separate connections for commands and data. Thus, the MitB attacker triggers two requests in the browser. The first request changes the state of the FTP server such that it opens a data port for the client to upload or download a file. Although the server returns the number of the data port to the client, that response is kept in the browser context of the targeted web server and is not accessible to the attacker. Thus, the attacker has to brute-force the correct data port on the server just as in a port-stealing attack [5]. Then the attacker triggers a second request in the browser and redirects it to the data port. For an upload attack, the full HTTP request, including any secret cookies in the header, is stored on the FTP server, where the attacker has read access. For a download attack, the attacker initially prepares a valid HTTP response with a malicious JavaScript payload and stores it on the FTP server. The response is returned to the client in the download attack.

Note that the two requests triggered by the MitB attacker do not use the same but different TLS connections, so FTP upload attacks work independently of connection reuse in the browser. For download attacks, the response by the FTP server on the data connection is free of any protocol noise, so it works in any browser regardless of content-sniffing.

**Email Upload Attacks.** SMTP and IMAP can be used to send or save emails to an attacker-controlled email account and thus are suitable for upload attacks. POP3 does not support uploading user data.

For SMTP, a MitB attacker can trigger a request in the browser to log into the attacker's account on the server and start submitting an email to that account. For IMAP, the request logs into the attacker's account on the IMAP server and saves a draft email to an attacker-controlled folder. In either case, the initial request prepares the server into a state where, if it receives a second browser request reusing the same connection, the content of the whole request (including the cookie in the header) would be exfiltrated to the attacker.

Note that the two requests triggered by the MitB attacker must use the same TLS connections, so email upload attacks require connection reuse in the browser.

**Email Download Attacks.** IMAP and POP3 can be used to download emails from an attacker-controlled email account and thus are suitable for download attacks. SMTP does not support downloading data.

For IMAP and POP3, a MitB attacker can trigger a request in the browser that contains commands to log into the attacker's account on the server, select a mailbox, and fetch an email containing the malicious payload previously stored there by the attacker. The response of the IMAP or POP3 server will contain the content of the whole email, including the email body with the malicious payload.

Note that the responses of the IMAP and POP3 servers include the whole transaction of the request, including the server banner, any error messages, and responses to the lo-

gin and other commands, which precede the content of the downloaded email. In practice, email download attacks only succeed if the victim browser supports content-sniffing.

## 6.2 Exploitability of Server Implementations

We identified popular SMTP, IMAP, POP3, and FTP implementations, based on an Internet-wide banner scan (see Table 7 in the appendix). We installed the most current version of these servers in a lab setting using the default configuration. Then we evaluated them for their exploitability in cross-protocol attacks against HTTPS by sending messages over the network and measuring the responses. We tested for:

1. Tolerance to HTTP request lines using the `POST` method, by sending the input string `POST / HTTP/1.1` as the *first* command to the server. If the server did not terminate the connection, it is considered tolerant.

2. Tolerance to HTTP header lines in the `key: value` format, by sending `Connection: keep-alive` as the *second* command to the server. If the server did not terminate the connection, it is considered tolerant.

3. The maximum number of syntax errors that are tolerated before the connection is terminated, by sending the same invalid command multiple times in a single session. If the server answered more than 100 invalid commands, we concluded that no limitation is implemented.

4. The availability of commands or error messages that allow reflected XSS attacks, by manual and tool-assisted exploration of the protocol syntax. We stopped searching when we found a reflection sufficient for a JavaScript exploit or when we exhausted the standard command list for the protocol (including popular extensions).

We did not separately evaluate download and upload attacks because these rely on standard protocol behavior that is already covered in the general description of these attacks.

## 6.3 Experimental Results

We evaluated 25 different application servers and their exploitability in cross-protocol attacks. Note that our list includes servers like Dovecot and Courier, which implement both IMAP and POP3. We count these servers twice as their implementations of these protocols have different properties with respect to cross-protocol attacks, as is apparent in our evaluation results. Our list also includes an old version of ProFTPD as a baseline test. Versions before 1.3.5e are known to be exploitable by the original cross-protocol attack on HTTPS by Jann Horn [38], while later versions were patched to detect these attacks. For each protocol and implementation, we verified which attack methods can be used with at least one browser to launch a cross-protocol attack on HTTPS. The

| | Server | HTTP Request Tolerant | HTTP Header Tolerant | Max. # of Errors | Reflects ASCII |
|---|---|---|---|---|---|
| SMTP | Postfix | ○ | ○ | 20 | ● |
| | Exim | ● | ● | 3 | ● |
| | Sendmail | ◐[a] | ● | 25 | ● |
| | MailEnable | ● | ● | 15[b] | ○ |
| | MDaemon | ● | ● | 3 | ● |
| | OpenSMTPD | ● | ● | ∞ | ● |
| IMAP | Dovecot | ● | ● | 3 | ◐[c] |
| | Courier | ● | ● | 10[d] | ● |
| | Exchange | ● | ● | 3 | ● |
| | Cyrus | ● | ● | ∞ | ● |
| | Kerio Connect | ● | ● | ∞ | ● |
| | Zimbra | ● | ● | ∞ | ● |
| POP3 | Dovecot | ● | ● | 3[d] | ◐[c] |
| | Courier | ● | ● | ∞ | ○ |
| | Exchange | ● | ● | 3 | ○ |
| | Cyrus | ● | ● | ∞ | ○ |
| | Kerio Connect | ● | ● | ∞ | ○ |
| | Zimbra | ● | ● | ∞[e] | ○ |
| FTP | Pure-FTPd | ○[f] | ● | ∞ | ● |
| | ProFTPD <1.3.5e | ● | ● | ∞ | ● |
| | ProFTPD ≥1.3.5e | ○ | ● | ∞ | ● |
| | Microsoft IIS | ● | ● | ∞ | ● |
| | vsftpd | ● | ● | ∞ | ◐[g] |
| | FileZilla Sever | ● | ● | ∞ | ● |
| | Serv-U | ● | ● | ∞ | ● |

● Favorable to attacker.
◐ Favorable to attacker with restrictions (see footnote).
○ No exploit found.
∞ No limit found. Tested with $> 100$ commands.
[a] Only with STARTTLS.
[b] Buffered commands are processed before connection is closed.
[c] Full XSS payload reflection post-auth.
[d] Counter is reset after valid command.
[e] 5 (with possible reset) after auth.
[f] Tolerant if compiled `--with-minimal`.
[g] Only post-auth with write permissions.

Table 2: HTTP header tolerance, error tolerance, and availability of commands suitable for reflected XSS in the evaluated application servers.

evaluation results are given in Table 2 and their exploitability for cross-protocol attacks is summarized in Table 3.

**SMTP.** All SMTP servers except Postfix and Sendmail were tolerant towards HTTP request and header lines. Postfix implements a detection for HTTP requests as well as HTTP headers. As soon as a command contains an HTTP status line or a key-value pair separated by a colon, the server will immediately terminate the connection. Sendmail only detects

HTTP requests at the very start of a connection. If STARTTLS is used, the first command inside the connection can be sent by the attacker, bypassing the detection.

Except OpenSMTPD, all tested SMTP implementations abort after a maximum number of errors. This is surprising, because the SMTP standard demands that a server *must not* close the connection in response to an unknown command [40]. A special case regarding the allowed maximum number of errors is MailEnable as it allows only 15 errors, but continues processing all remaining buffered commands before terminating the connection.

All SMTP servers except MailEnable allowed at least one XSS reflection vector.

**IMAP.** All IMAP servers were tolerant towards HTTP request and header lines.

With regards to the number of allowed errors, Courier implements a counter that resets after receiving a valid command. In this evaluation, we mark Courier as not sufficiently error tolerant to allow cross-protocol attacks. However, a more sophisticated attacker might be able to bypass this by inserting valid commands disguised as HTTP headers in the request, resetting the error counter. The inserted header must be CORS-safe [46] in order to avoid a preflight request sent by the browser. An example header to reset the error counter would be `Accept: noop`, where `Accept:` is interpreted as the IMAP tag and `noop` as the IMAP command. We did not evaluate if the attacker can position such a header within the first ten lines of the request in common browsers.

Zimbra (POP3) allows unlimited errors but only before authentication. Post-authentication, the server will terminate after five errors. As authentication in cross-protocol attacks occurs within the body of the HTTP request, which is completely controlled by the attacker, no errors are expected after authentication due to protocol noise. Therefore, this does not affect the attack. Of the other IMAP servers, only Exchange enforced a limit on the number of errors. With Dovecot IMAP, a full XSS payload reflection is only possible post-auth using the command SELECT or STATUS for IMAP. All other IMAP servers had pre-auth commands usable for reflection.

**POP3.** All POP3 servers were tolerant towards HTTP request and header lines.

Dovecot POP3 implements a counter that resets after valid commands. However, with only three consecutive errors allowed, and the restrictions of the POP3 protocol, it seems highly unlikely that an attacker can bypass the error limit by inserting attacker-controlled header lines to reset the counter.

As for reflection, Dovecot POP3 allows XSS reflection only post-authentication using an unknown command. For all other POP3 servers we could not find any reflection vectors.

**FTP.** Two FTP servers could detect and block HTTP requests. If a command to these servers starts with an HTTP status line, the servers immediately close the connection.

| | Server | Attack Method | | | # HTTPS |
|---|---|---|---|---|---|
| | | Upload | Download | Reflection | |
| SMTP | Postfix | ○[a] | - | ○[b] | |
| | Exim | ○[a] | - | ○[b] | |
| | Sendmail | ○[a] | - | ◐[e] | 11,365 |
| | MailEnable | ○[a] | - | ○ | |
| | MDaemon | ○[a] | - | ○[b] | |
| | OpenSMTPD | ○[a] | - | ○[c] | |
| IMAP | Dovecot | ○[a] | ○[b] | ○[b] | |
| | Courier | ○[a] | ○[b] | ○[b] | |
| | Exchange | ○[a] | ○[b] | ○[b] | |
| | Cyrus | ○[a] | ● | ● | 14,029 |
| | Kerio Connect | ○[a] | ● | ● | 7,852 |
| | Zimbra | ○[a] | ● | ● | 9,578 |
| POP3 | Dovecot | - | ○[b] | ○[b] | |
| | Courier | - | ● | ○ | 30,759 |
| | Exchange | - | ○[b] | ○ | |
| | Cyrus | - | ● | ○ | 9,079 |
| | Kerio Connect | - | ● | ○ | 4,501 |
| | Zimbra | - | ● | ○ | 7,927 |
| FTP | Pure-FTPd | ○[d] | ○[d] | ○[d] | |
| | ProFTPD <1.3.5e | ■ | ■ | ● | 13,481 |
| | ProFTPD ≥1.3.5e | ○[d] | ○[d] | ○[d] | |
| | Microsoft IIS | ■ | ■ | ● | 19,817 |
| | vsftpd | ■ | ■ | ◐[f] | 7,211 |
| | FileZilla Server | ■ | ■ | ● | 1,555 |
| | Serv-U | ■ | ■ | ● | 1,429 |
| | **Total Unique** | | | | 114,197 |

■ Exploitable in *all* browsers.
● Exploitable with content sniffing (IE, Edge Legacy).
◐ Exploitable with content sniffing (IE, Edge Legacy), with some limitations described in footnote.
○ No exploit found.
- Attack method not applicable.
[a] Not exploitable because no browser reuses the connection.
[b] Not exploitable due to too many errors with all browsers.
[c] Not exploitable due to lack of command pipelining.
[d] Not exploitable due to HTTP detection.
[e] Exploitable on all ports except 465 (implicit TLS).
[f] Exploitable if attacker can login with write permission.

Table 3: Summary of our evaluation of application servers for each attack method. The last column shows the number of affected HTTP servers from our scan (see Subsection 7.2).

All FTP servers tolerated an arbitrary number of errors without terminating the connection.

All FTP servers, except vsftpd, had at least one command that allowed reflection before authentication. In vsftpd, a user with write permission can reflect ASCII post-authentication using the command `MKDIR`.

**Exploitability of Servers.** We now summarize our results with respect to each attack scenario and browser (see Table 3).

Five FTP servers were exploitable in an upload attack with any browser. Two secure servers, Pure-FTPD and ProFTPD, were able to detect HTTP headers and thus prevent all HTTP cross-protocol attacks. For SMTP and IMAP upload attacks, no tested server was exploitable because the server response is not a valid HTTP/1.1 response line. As discussed in the previous section, the missing HTTP/1.1 response line prevents the browsers from reusing the TLS connection.

Three IMAP servers and four POP3 servers were exploitable in a download attack with a browser supporting content-sniffing. Five FTP servers were exploitable in a download attack with any browser. Secure IMAP and POP3 servers were not error tolerant enough to allow a successful attack.

Nine application servers were exploitable in a reflection attack with a browser supporting content-sniffing. Two out of these servers were exploitable with some technical restrictions: Sendmail was only exploitable on a STARTTLS port, and vsFTPd requires an attacker having write permission on the FTP server. Servers that were not exploitable in a reflection attack either blocked HTTP requests (Postfix and ProFTPD $\geq$ 1.3.5), were not sufficiently error tolerant, or did not provide a reflection vector in any of the tested commands. As a special case, OpenSMTPD is not tolerant to HTTPS requests or other batched multi-line input due to missing support for command pipelining.

In total, 13 of 25 evaluated application servers can be exploited with at least one attack method in at least one browser.

### 6.4 Lab Setup

To demonstrate that the exploitable application servers actually can be used in fully working attacks on HTTPS, we created a lab setup containing proof-of-concepts for each protocol and attack method from our evaluation. In particular, the lab contains Sendmail 8.16.1 (SMTP), Cyrus 2.4.17 (IMAP), Courier 1.0.6-1 (POP3), and vsftpd 3.0.2 (FTP). The lab is based on the containerization software Docker, and contains an attack server and a web server with a website to attack. We configured publicly reachable domains and valid certificates from Let's Encrypt and implemented an actual MitM attacker capable of relaying and altering traffic as well as injecting additional packages. We then implemented working FTP download and upload attacks on vsftpd for all browsers, as well as reflection and download attacks for Cyrus IMAP, a download attack for Courier POP3, and a reflection attack for Sendmail in combination with Edge Legacy and IE. The attacks are implemented in Python. The complete lab setup, including all proof of concept attacks, is available on Github.[3]

[3] https://github.com/RUB-NDS/alpaca-code

## 7 Large Scale TLS Study

We evaluated the number of HTTPS servers which are vulnerable to cross-protocol attacks with SMTP, IMAP, POP3, or FTP in an internet-wide scan of the IPv4 address space by looking for servers with trusted, compatible certificates. Additionally, we analyzed how these servers react to invalid server names with SNI and how they react if they cannot choose a valid application layer protocol with ALPN.

### 7.1 Methodology

In order to evaluate how many application servers have trusted certificates compatible with HTTPS servers, we conducted multiple IPv4 scans on standard and well-known application ports for SMTP (25, 587, 465, 26, 2525), IMAP (143, 993), POP3 (110, 995), and FTP (21, 990) between July and October 2020, using ZMap [21] and ZGrab 2.0.[4] We excluded all hosts that could not complete a TLS handshake. We then determined which of these servers have a trust path to a generally trusted root CA. We considered a CA as generally trusted if it is trusted by either Mozilla, Google, Microsoft, Apple, Oracle, or OpenJDK. We then gathered all trusted certificates and extracted their Common Names (CN) and Subject Alternative Names (SAN) in order to find corresponding HTTPS servers. For entries that contained a `*`, we guessed the subdomain by replacing `*` with `www`. We then tried to connect to these hostnames on port 443 using the HTTPS protocol and collected the presented certificates.

We performed two more scans on those SMTP, IMAP, POP3, and FTP application servers that offered a trusted certificate. In the first scan, we estimated the number of application servers that tolerate incorrect SNI hostnames by performing a TLS handshake with the SNI hostname `example.com`. We recorded if the TLS handshake completes successfully despite the mismatching hostname. In the second scan, we estimated the number of application servers that tolerate incorrect application layer protocols by performing a TLS handshake with the same ALPN extension as sent by the Chrome web browser. We recorded if the TLS handshake completes successfully despite the mismatching protocol identifiers.

### 7.2 Results of Internet-Wide Scans

The results of our scans can be seen in Table 4. Across all protocols, 62,85% of the discovered TLS application servers used generally trusted certificates. A notable outlier is FTP on port 21, where the number of trusted certificates was only 44%. A possible explanation is that FTP server certificates are often signed by private CAs that are not generally trusted by browsers. We found that about 25% of the untrusted FTP certificates were signed by such private CAs.

[4] https://github.com/zmap/zgrab2

| Protocol | Port | STARTTLS | Server IPs with TLS | | Certificate Names (CN & SAN) | |
|---|---|---|---|---|---|---|
| | | | Total | Valid Certificate | # Unique | # HTTPS |
| SMTP | 25 | Yes | 3,427,465 | 1,744,052 (50,88%) | 1,048,090 | 782,710 (74.68%) |
| SMTP | 587 | Yes | 3,495,626 | 2,471,893 (70,71%) | 1,176,374 | 821,534 (69.85%) |
| SMTPS | 465 | - | 3,511,544 | 2,450,062 (69,77%) | 1,046,240 | 724,557 (69.27%) |
| SMTP | 26 | Yes | 565,672 | 514,425 (90,94%) | 130,624 | 79,234 (60.66%) |
| SMTP | 2525 | Yes | 231,009 | 139,536 (60,40%) | 50,514 | 31,009 (61.40%) |
| IMAP | 143 | Yes | 3,707,577 | 2,463,293 (66,44%) | 1,103,455 | 782,410 (70.92%) |
| IMAPS | 993 | - | 3,919,999 | 2,597,232 (66,26%) | 1,287,370 | 926,313 (71.97%) |
| POP3 | 110 | Yes | 3,551,226 | 2,342,545 (65,96%) | 983,912 | 690,111 (70.15%) |
| POP3S | 995 | - | 3,828,411 | 2,580,379 (67,40%) | 1,170,197 | 848,744 (72.56%) |
| FTP | 21 | Yes | 4,826,891 | 2,130,271 (44,13%) | 675,432 | 421,923 (62.48%) |
| FTPS | 990 | - | 305,646 | 282,382 (92,39%) | 115,292 | 95,197 (62.73%) |
| **Total** | | | 31,371,066 | 19,716,070 (62,85%) | 2,088,328 | 1,441,628 (69.03%) |

Table 4: Results from our internet-wide scan by protocol and port (July to October 2020). We first give the number of IP addresses that provide the given service and allow a successful TLS handshake to be made. Then we show the number of those IP addressses that offer a certificate that is considered valid for a browser (except for hostname matching). Next we give the number of unique names found in the CN and SAN of the valid certificates. Finally, we give the number of HTTPS servers we found among these names, with * replaced by `www` as the most common guess for web servers using wildcard certificates.

**TLS Version.** Previous studies analyzing the TLS ecosystem have shown that servers running SMTP, IMAP, or FTP do not offer timely TLS protocol support [6, 36, 44]. Running a service with outdated TLS versions can negatively affect the cross-protocol attack execution because current browsers support TLS 1.2 and TLS 1.3 only.

Our scan does not include servers supporting only TLS 1.3 due to lack of support in the version of ZGrab we used, but we suspect that the number of such exclusive servers is marginal among the long-established protocols we analyzed. Our scans show that across all analyzed protocols, 90% to 96% of the scanned application servers with trusted certificates support TLS 1.2, while the rest only support older versions. This means that successful attack exploitation can fail in at most 10% of these servers due to missing support for TLS 1.2.

**ALPN and SNI.** We removed all host responses from the data set for which the handshake was either successful or could be attributed to an unrelated error (such as connection timeout), and were left with a marginal number of hosts which potentially rejected the TLS handshake because of the ALPN or SNI extension. Depending on the protocol and port, we can give an upper bound for servers potentially supporting ALPN or SNI correctly below 0.5%. We conclude that ALPN or SNI do not pose an obstacle to cross-protocol attacks today.

**Web Servers Vulnerable to Cross-Protocol Attacks.** Across all analyzed protocols, we collected a total number of 2,088,328 distinct hostnames. Our search for HTTPS servers on those hostnames revealed a total of 1,441,628 HTTPS servers for which at least one SMTP, POP3, IMAP or FTP server exists that was using a generally trusted certificate, which is 69% of all the unique hostnames scanned. Of these web servers, 24,202 were in the Tranco 1M list [42] of the most prominent hosts on the Internet.[5] This means that for the majority of the servers with trusted certificates on SMTP, POP3, IMAP, or FTP, there exists an HTTPS server with a compatible certificate vulnerable to a general TLS cross-protocol attack, where application data is processed by the substitute server rather than the intended web server.

**Vulnerable Web Servers Paired With Exploitable Application Server.** Based on our banner scan (see Appendix A), we counted all unique web servers (among the 1.4M candidates) for which we could identify at least one application server that was exploitable in our lab setting (see rightmost column of Table 3 in Section 6). Sometimes the same web server is exploitable by several application servers (e.g., IMAP and POP3), so the total number of unique web servers is smaller than the sum over all protocols.

In total, we found 114,197 unique web server hostnames that can be attacked using an exploitable SMTP, IMAP, POP3, or FTP server with a trusted and compatible certificate.

## 8 Cross-Protocol Attacks without MitM

So far, we have assumed the scenario of an active MitM, which is a reasonable attacker model for attacks on TLS. In this section, we discuss necessary conditions in order to adapt the presented attacks to a pure MitB scenario. In this attacker model, the attacker forces the victim browser to send a HTTP

[5] Downloaded on 2020-10-11.

| | Chrome | Firefox | IE | Edge Legacy | Edge | Opera | Safari |
|---|---|---|---|---|---|---|---|
| SMTPS (465) | ○ | ○ | ● | ● | ○ | ○ | ○ |
| IMAPS (993) | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| POP3S (995) | ○ | ○ | ● | ● | ○ | ○ | ○ |
| FTPS (990) | ● | ● | ● | ● | ● | ● | ● |

○ Port blocked ● Port not blocked

Table 5: Some ports available to a MitB attacker by browser.

| | | Chrome | Firefox | IE | Edge Legacy | Edge | Opera | Safari |
|---|---|---|---|---|---|---|---|---|
| same domain, different port | DOM access | ○ | ○ | ● | ● | ○ | ○ | ○ |
| | Get cookie | ● | ● | ● | ● | ● | ● | ● |
| | Set cookie | ● | ● | ● | ● | ● | ● | ● |
| sub domain, different port | DOM access | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| | Get cookie | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ |
| | Set cookie | ● | ● | ● | ● | ● | ● | ● |

● access blocked ○ access denied ◐ cookie dependent

Table 6: SOP interpretation in different browsers.

request directly to a different application server that, as far as the browser is concerned, belongs to the same security context as the targeted web server based on the origin of the request. Besides requiring a weaker attacker model, this attack method offers several advantages: 1. Certificate cross-compatibility is no longer a requirement because the client validates the server's actual certificate. 2. SNI no longer influences the attack, as the extension contains the correct server name.

However, in a MitB-only setting, where the attacker can no longer redirect traffic to a different hostname and port other than the intended as seen by the browser, several web-related restrictions exist, namely lack of STARTTLS support, port blocking in the browser, and the Same-Origin Policy.

**STARTTLS Usage.** Because a pure MitB attacker cannot upgrade a non-TLS connection to a secure one (by sending a `STARTTLS` command), the attack can only work with servers using implicit TLS (SMTPS, IMAPS, POP3S, FTPS).

**Port Blocking.** As a workaround to counter early cross-protocol attacks [13, 59] in 2001, browsers block access to specific well-known ports. An excerpt is given in Table 5.

As expected, most browsers block access to ports used by SMTP, IMAP, POP3, and FTP, with some exceptions. For example, port 990 is not blocked in any tested browsers, so cross-protocol attacks exploiting FTPS are still possible. Furthermore, Edge Legacy and Internet Explorer do not block port 465 (SMTPS) and 995 (POP3S), allowing cross-protocol attacks exploiting these services. On the other hand, these browsers do block access to the plaintext (non-TLS) variants of these protocols at port 25 (SMTP) and 110 (POP3).

Therefore, due to port blocking, *most* of our attacks do not work in a pure web attacker scenario *unless* a service runs on a non-standard port. In practice, this is not unrealistic, as services are frequently deployed on non-standard ports for a variety of administrative reasons.

**Same-Origin Policy.** Another obstacle to deal with in a pure web attacker model is cross-site limitations due to the Same-Origin Policy (SOP) [58, 61], including cookie policies (see Table 6). DOM access from one origin (identified by a `protocol://host:port` tuple) to a different origin is not allowed. However, Edge Legacy and Internet Explorer ignore port numbers in the SOP. For example, `host:995` (POP3S) has access to `host:443`, thereby allowing DOM manipulation (i.e., reading or writing website content, inserting script tags, etc.). Furthermore, technologies such as CORS [32] exist to punch holes into the SOP. According to Müller [48], around 0.15% of the Alexa Top 1M websites are misconfigured to allow cross-site requests with session cookies.

The SOP is more lax when it comes to cookies. As specified in RFC 6265 [7], cookies are *not* port-dependent. For example, `host:995` can access the cookies for `host:443` in all tested browsers. Read access to cookies for subdomains is only possible if the `Domain` flag is explicitly set. Cahn et al. [11] crawled the Alexa Top 100k websites and found that this is the case for 81.5% of the cookies, including all subdomains. However, their work did not focus on session cookies, where the numbers may be lower. Policies for setting cookies are even less strict, as a subdomain is allowed to set a cookie for the top domain. For example, `pop3.host:995` can set a cookie for `host:443` in all browsers, which can lead to session fixation attacks [41], where the attacker locks the user into a session already controlled by the atacker before the user even logs in.

## 8.1 Practical Example

As a proof of concept demonstration, we registered an account at Mailfence, a security-focused email provider. As MitB, we posted HTML form data to `https://mailfence.com:995` to log into our account and retrieve the content of an HTML email (download attack), resulting in JavaScript execution in the context of `https://mailfence.com` for browsers that ignore the port number in the SOP, such as Internet Explorer. The issue was acknowledged by the vendor as stored XSS. We found similar exploitable issues in the MitB attacker model, in a major bitcoin exchange, the website of a large university, and the Government of India's webmail service.[6]

[6] All tested services encourage researchers to search for security bugs and we followed their requirements for responsible disclosure.

## 9 Countermeasures

**Countermeasures at the Application Layer.** Previous efforts to stop cross-protocol attacks tried to mitigate the issue at the application layer, for example, by closing the connection if HTTP is detected instead of a valid command. From a practical point of view it is unreasonable to expect implementers to be aware of all (including future) possible cross-protocol attacks and defend against them one by one.

While such measures can potentially stop the exploitation of individual protocol confusions, they cannot stop the attack in general. Whenever a client finishes the handshake with $S_{sub}$, the authentication as promised by TLS has already been broken. At this point, no application data has been exchanged yet, therefore no application layer countermeasure can prevent the general cross-protocol attack.

**Countermeasures with TLS Certificates.** A common proposal is to use different (incompatible) certificates for different service endpoints. However, enforcing such a policy is challenging in practice. Certificate validation is limited to hostnames, and thus each service would have to be hosted on a unique subdomain. Furthermore, no certificate should be issued for more than one hostname, which effectively prohibits the use of wildcard certificates. However, the very common use of wildcard certificates in practice shows that they provide significant value to administrators. Even strict certificate exclusivity does not prevent all possible attacks. The attacker could still steal the cookie using a service hosted on a subdomain or perform session fixation attacks (see Section 8).

Another idea would be to define different certificate usages for distinct services. While the X.509 standard defines the extended key usage extension [14], this extension only allows to distinguish TLS server certificates from those used for email signing, IPsec, or OCSP, and does not provide a mechanism to authenticate the application protocol on top of TLS.

We conclude that the required organizational and behavioral changes to achieve certificate exclusivity are so large that they can only be considered a long term countermeasure.

**ALPN Mitigates All Cross-Protocol Attacks.**

In 2015, Horn suggested the use of ALPN by protocol designers to mitigate cross-protocol attacks. We now describe how this countermeasure can be implemented in a backwards compatible way. If ALPN is supported by both client and server, the standard requires that the connection is closed if no common protocol can be negotiated. This strict implementation mitigates all cross-protocol attacks, because a client and the substitute server implementing a different protocol than the client will never complete a TLS handshake.

Today, we see different levels of ALPN support deployed. For HTTPS, all major clients already implement ALPN to support HTTP/2, so deploying ALPN in exploitable application servers will prevent our attacks on HTTPS. In an internet-wide scan we found that 72.5% of HTTPS servers already support ALPN. Although this is promising, we also found that less than 1.3% terminate the connection if no protocol can be negotiated. We have also shown in our scans that virtually all SMTP, IMAP, POP3, and FTP servers do not support ALPN or do not terminate if no protocol can be negotiated.

As a path forward, we propose that initially servers start to implement ALPN strictly according to the standard, so connections created by clients sending the ALPN extension (i.e., browsers) are protected from exploitation. In parallel, clients for all protocols (SMTP, IMAP, POP3, and FTP) can be upgraded to send the ALPN extensions. Migrating to this secure configuration is easy and backwards-compatible, as the clients and servers can independently enable the extension on their respective side at some convenient time, while still accepting legacy connections. Once a client and an application server have *both* enabled ALPN, that particular server can no longer be exploited to attack connections by that client to other, vulnerable servers in the network.

Eventually, clients and servers may choose to require the use of ALPN by the other side, at the cost of breaking backwards compatibility with legacy implementations.

**Countermeasures with SNI.** The same way the ALPN extension protects against cross-protocol attacks, the SNI extension can protect against cross-hostname attacks, if it is implemented strictly (ie., the connection is terminated if no matching host is found), which is allowed by the standard. This can protect against cross-protocol attacks where the intended and substitute server have different hostnames, but also against some same-protocol attacks such as HTTPS virtual host confusion [16] or context confusion attacks [65].

Unfortunately, some servers are currently not entirely aware of the hostnames they are responsible for. Adding a strict SNI validation to those servers can cause connections to break if hostnames are missing or clients are misconfigured. Still, we recommend enabling strict SNI checking if possible, in particular for new configurations.

**Same-Host, Same-Protocol, Cross-Port Attacks.** Even with strict ALPN and SNI implementations, we still face potential confusion attacks when the intended and substitution server have the same hostname, implement the same protocol, but run on different ports. These *cross-port* attacks can currently not yet be mitigated at the TLS layer, because there is no way for the client to communicate the intended port number to the server. Defining such a feature, for example as a new TLS extension, is certainly possible, but would require an upgrade to all TLS libraries and applications.

## 10 Related Work

Early cross-protocol attacks were found in cryptographic systems. Kelsey, Schneier and Wagner [39] described how new protocols can be designed to allow such attacks on existing

protocols, forshading some of the problems occurring when key material, certificates, or cryptographic protocols (such as TLS) are reused for different applications. They also gave basic principles for protocol design to avoid such issues. Their work was expanded by Canetti et al. [12], who considered the environmental requirements for authentication protocols and showed that even strong protocols can fail for external reasons. For TLS, cryptographic cross-protocol attacks were examined in [62], [43] and [6] (DROWN). Nir and Gueron [18] analyzed a similar weakness in TLS 1.3 PSK where the client is running a server with the same pre-shared key as the intended server. In their "Selfie" attack, the attacker redirects the messages from the client $C$ back to its own server without the client noticing that confusion.

A first example for an application layer cross-protocol attack was described by Topf [59], who showed how to send emails via SMTP over HTTP from an HTML form. He recognized this as a way to access intranet services behind a firewall. A first systematization of these attacks was provided by Alcorn [1] and subsequently extended to demonstrate the impact on internal networks behind firewalls in [2] (*inter-protocol exploitation*) by giving an attack on the Asterisk Manager Interface through HTTP. Other authors applied these techniques to attack UPnP [31], FTP [54], IMAP [50], and Redis [33] servers in internal networks, often paired with other vulnerability exploits to achieve remote code execution. Other works discussed how to to send commands to network printers [63] or spam [34] from HTML websites. The most recent summary was given by Prynn [53] who also showed how HTTP can be combined with binary protocols. Common to these works is that they give no consideration to TLS and that they are attacking the protocol wrapped inside the HTTP protocol, rather than the HTTP server. These attacks were also considered in the design of HTTP/2 [9].

A simple XSS attack against web servers with colocated services vulnerable to reflection attacks was described by Gauci [29, 30]. The first structured presentation of the XSS attack scenario was presented by Alcorn [1]. Horn presented a first example for a JavaScript download attack using FTP over HTTPS as a MitM attacker in a bug report against the *ProFTPD* FTP sever [37], which is the first time that TLS application data confusion was used to enable cross-protocol attacks. Horn also pointed out a vulnerability in vsftpd and a potential vulnerability in Dovecot IMAP, and suggested the use of ALPN to mitigate the attacks [38]. In their study on printer attacks, Müller et al. showed how to use cross-site printing to spoof CORS headers and thus get access to data from a different origin [49].

Another line of research considered attacks on TLS application data confusion within the same protocol, rather than different protocols. Delignat-Lavaud and Bhargavan [16] analyzed how a MitM can exploit HTTPS virtual hosting configurations, and Zhang et al. [65] found even more HTTPS MitM attacks, exploiting insecure web security policies in the substitute server or mixing TLS with plaintext content.

While our attacks require a strong attacker with MitM and MitB capabilities, such attacker model is typical when targeting the TLS protocol. The scenario was first described by Rizzo and Duong in their BEAST attack to exploit predictable CBC (cipher block chaining) initialization vectors in TLS 1.0 [19]. In order to retrieve secret data, the BEAST attacker runs a JavaScript in victim's browser to trigger carefully created requests to the server. The attacker observes the encrypted requests, whose structure leaks information about the secret data. This attacker model was later used in CRIME [20], Lucky 13 [4], POODLE [45], and attacks on RC4 [3,28,60]. In Lucky 13 and POODLE, the MitM attacker also actively modifies the TLS traffic as in our attack.

## 11 Conclusions

We demonstrated that the lack of strong authentication of service endpoints in TLS can be abused by attackers to perform powerful cross-protocol attacks with unforeseeable consequences. Our internet-wide scans showed that it is common for administrators to deploy compatible certificates across multiple services, possibly without consideration to cross-protocol attacks. We also showed that cross-protocol attacks are practical, although the impact is limited and difficult to assess from lab experiments alone. In the real-world, cross-protocol attacks will always be situational and target individual users or groups. However, it is also clear that existing countermeasures are ineffective because they do not address all possible attack scenarios.

We have identified one countermeasure that is far superior to others: the pervasive use of the ALPN extension to TLS by both client and server. Luckily, ALPN is easy to deploy with the next software update without affecting legacy clients or servers.

In a broader sense, this work demonstrates yet again that all cryptographic measures, when applied to real world applications, should be bound to the context of their legitimate use to prevent confusion attacks on the protected content. Binding the TLS connection to a specific application layer protocol allows peers to protect themselves against any known and unknown cross-protocol attack. With SNI, this protection can be extended to same-protocol attacks on different hostnames. These countermeasures make sure that a message for the intended protocol is not mistaken for a message in the substituted protocol, as demanded by the Rule of Thumb 5 in [12] for safeguarding authentication protocols against environmental threats. However, we have also seen that services that share the same hostname and protocol can not be protected against confusion attacks by existing TLS standards.

Our work can be extended in different directions. We have only studied the vulnerability of HTTPS to cross-protocol attacks based on FTP and email protocols. Other cross-protocol attack scenarios and protocol combinations need to be ana-

lyzed. This does not only include text-based protocols, as similar cross-protocol attacks can be applicable to binary protocols as well. Our attacks work because of the lack of authentication between TLS and the application layer protocols. Similar problems can arise in other cryptographic protocols, such as DTLS [57] or IPsec [25].


## Acknowledgements

Marcus Brinkmann was supported by the German Federal Ministry of Economics and Technology (BMWi) project "Industrie 4.0 Recht-Testbed" (13I40V002C). Robert Merget was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. We also thank our shepherd Zakir Durumeric as well as our anonymous reviewers for improving the final version of the paper.


## References


[1] Wade Alcorn. Inter-protocol communication, 2006. https://web.archive.org/web/20111229080404/http://www.bindshell.net/papers/ipc.html (accessed 2019-06-26).

[2] Wade Alcorn. Inter-process exploitation, 2007. https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/inter-protocol_exploitation.pdf (accessed 2019-06-26).

[3] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of rc4 in TLS. In Samuel Talmadge King, editor, *22nd USENIX Security Symposium (USENIX Security 13)*, pages 305–320, Washington D.C., USA, August 14–16, 2013. USENIX Association.

[4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *IEEE Symposium on Security and Privacy*, 0:526–540, 2013.

[5] M. Allman and S. Ostermann. FTP Security Considerations. RFC 2577 (Informational), May 1999.

[6] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, August 2016. USENIX Association.

[7] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[8] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *2009 30th IEEE Symposium on Security and Privacy*, pages 360–371, 2009.

[9] M. Belshe, R. Peon, and M. Thomson (Ed.). Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[10] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[11] Aaron Cahn, Scott Alfeld, Paul Barford, and Shanmugavelayutham Muthukrishnan. An empirical study of web cookies. In *Proceedings of the 25th International Conference on World Wide Web*, pages 891–901, 2016.

[12] Ran Canetti, Catherine Meadows, and Paul Syverson. Environmental Requirements for Authentication Protocols. In *Software Security — Theories and Systems*, pages 339–355. Springer Berlin Heidelberg, 2003.

[13] CERT Coordination Center. Vulnerability note vu#476267, 2001.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[15] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard), March 2003.

[16] Antoine Delignat-Lavaud and Karthikeyan Bhargavan. Network-based Origin Confusion Attacks against HTTPS Virtual Hosting. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, may 2015.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[18] Nir Drucker and Shay Gueron. Selfie: reflections on TLS 1.3 with PSK. Cryptology ePrint Archive, Report 2019/347, 2019. https://eprint.iacr.org/2019/347.

[19] Thai Duong and Juliano Rizzo. Here come the $\oplus$ Ninjas. Ekoparty security conference, 2011.

[20] Thai Duong and Juliano Rizzo. The crime attack. Ekoparty security conference, 2012.

[21] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 605–620, Washington, D.C., August 2013. USENIX Association.

[22] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), January 2011.

[23] Chris Evans. vsFTPd-2.1.0 released, 2009. https://scarybeastsecurity.blogspot.com/2009/02/vsftpd-210-released.html (accessed 2020-10-15).

[24] R. Fielding (Ed.) and J. Reschke (Ed.). Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[25] S. Frankel and S. Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071 (Informational), February 2011.

[26] N. Freed. SMTP Service Extension for Command Pipelining. RFC 2920 (Internet Standard), September 2000.

[27] S. Friedl, A. Popov, A. Langley, and E. Stephan. Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (Proposed Standard), July 2014.

[28] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. Attacks only get better: Password recovery attacks against rc4 in TLS. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium (USENIX Security 15)*, pages 113–128, Washington D.C., USA, August 12–14, 2015. USENIX Association.

[29] Sandro Gauci. Extended HTML Form Attack, 2002. https://eyeonsecurity.org/papers/extendedformattack.html (accessed 2019-10-18).

[30] Sandro Gauci. The Extended HTML Form attack revisited, 2008. https://dl.packetstormsecurity.net/papers/web/the-extended-html-form-attack-revisited.pdf (accessed 2020-09-26).

[31] Gnucitizen.org. Hacking the interwebs, 2008. https://www.gnucitizen.org/blog/hacking-the-interwebs/ (accessed 2019-07-09).

[32] W3C Web Hypertext Application Technology Working Group et al. CORS Protocol, 2018.

[33] Nicolas Grégoire. Trying to hack Redis via HTTP requests, 2014. https://www.agarri.fr/blog/archives/2014/09/11/trying_to_hack_redis_via_http_requests/index.html (accessed 2019-10-14).

[34] Robert Hansen. Javascript spam, 2007. http://web.archive.org/web/20090913204859/http://ha.ckers.org/blog/20070325/javascript-spam.

[35] P. Hoffman. SMTP Service Extension for Secure SMTP over TLS. RFC 2487 (Proposed Standard), January 1999.

[36] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kâafar. TLS in the wild: an internet-wide analysis of TLS-based protocols for electronic communication. In *NDSS 2016*, pages 1–15, 2016. Network and Distributed System Security Symposium 2016, NDSS'16 ; Conference date: 21-02-2016 Through 24-02-2016.

[37] Jann Horn. HTTPS/FTPS protocol confusion leads to XSS (ProFTP Bug 4143), 2014. http://bugs.proftpd.org/show_bug.cgi?id=4143#c0 (accessed 2019-06-26).

[38] Jann Horn. Two cross-protocol MitM attacks on browsers, 2015. https://var.thejh.net/http_ftp_cross_protocol_mitm_attacks.pdf (accessed 2020-08-27).

[39] John Kelsey, Bruce Schneier, and David Wagner. Protocol interactions and the chosen protocol attack. In *Security Protocols*, pages 91–104. Springer Berlin Heidelberg, 1998.

[40] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008.

[41] Mitja Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, 7, 2002.

[42] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[43] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A Cross-protocol Attack on the TLS Protocol. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 62–72, New York, NY, USA, 2012. ACM.

[44] Wilfried Mayer, Aaron Zauner, Martin Schmiedecker, and Markus Huber. No need for black chambers: Testing TLS in the e-mail ecosystem at large. *CoRR*, abs/1510.08646, 2015.

[45] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf.

[46] Mozilla MDN. Cross-Origin Resource Sharing (CORS) - Simple Requests, 2020.

[47] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939 (Internet Standard), May 1996.

[48] J. Müller. CORS misconfigurations on a large scale, 2017. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html (accessed 2021-02-09).

[49] J. Müller, V. Mladenov, J. Somorovsky, and J. Schwenk. Sok: Exploiting network printers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 213–230, 2017.

[50] Michele Orrù. Revitalizing the Inter-Protocol Exploitation with BeEF Bind, 2012. https://blog.beefproject.com/2012/11/revitalizing-inter-protocol.html (accessed 2019-07-09).

[51] J. Postel. DoD standard Transmission Control Protocol. RFC 761 (Historic), January 1980.

[52] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Internet Standard), October 1985.

[53] Tanner Prynn. Cross-protocol request forgery, 2018. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/cprf-1.pdf (accessed 2019-10-14).

[54] Antonio Quina. Inter-protocol communication - exploitation, 2012. https://www.secforce.com/blog/2012/11/inter-protocol-communication/ (accessed 2019-07-09).

[55] Marsh Ray and Steve Dispensa. Renegotiating TLS, 2009. https://web.archive.org/web/20091122081325/https://extendedsubset.com/Renegotiating_TLS.pdf (accessed 2021-02-09).

[56] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[57] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.

[58] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, 2017.

[59] Jochen Topf. The HTML form protocol attack, 2001. Published on the Bugtraq mailing list on 2001-08-15. https://www.jochentopf.com/hfpa/hfpa.pdf (accessed 2019-10-18).

[60] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking rc4 in wpa-tkip and TLS. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium (USENIX Security 15)*, pages 97–112, Washington D.C., USA, August 12–14, 2015. USENIX Association.

[61] W3C. Same-Origin Policy, 2010. https://www.w3.org/Security/wiki/Same_Origin_Policy (accessed 2021-02-09).

[62] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[63] Aaron Weaver. Cross-site printing, 2007. http://web.archive.org/web/20090919174421/http://www.net-security.org/dl/articles/CrossSitePrinting.pdf.

[64] Mike West. Content Security Policy Level 3, 2018. https://www.w3.org/TR/CSP3/ (accessed 2021-02-09).

[65] Mingming Zhang, Xiaofeng Zheng, Kaiwen Shen, Ziqiao Kong, Chaoyi Lu, Yu Wang, Haixin Duan, Shuang Hao, Baojun Liu, and Min Yang. Talking with familiar strangers: An empirical study on https context confusion attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2020.

## A Application Server Banner Scan

During our Internet-wide scan, we collected all banners from those application servers that could complete a TLS handshake. We used this data to identify the vendor of the deployed application server. In each banner, we replaced all non-letters by whitespace, lowercased all letters, and split at the word boundary. Then we replaced important multi-word sequences by single tokens, removed all single-letter tokens and removed tokens based on a stop list. We iterated manually through the most frequent tokens, and either assigned them to an implementation or to the stop list, until the identification was stabilized based on a 100k test set. The results are shown in Table 7.

| Pos. | SMTP (25) | SMTP (587) | SMTPS (465) | SMTP (26) | SMTP (2525) |
|---|---|---|---|---|---|
| 1. | Postfix<br>35.58% (1,219,598) | Exim<br>56.13% (1,962,165) | Exim<br>56.82% (1,995,277) | Exim<br>96.07% (543,451) | Exim<br>60.80% (140,457) |
| 2. | (unknown)<br>27.15% (930,442) | Postfix<br>17.65% (617,110) | (unknown)<br>18.86% (662,308) | (unknown)<br>2.01% (11,397) | (unknown)<br>26.78% (61,870) |
| 3. | Exim<br>21.94% (752,034) | (unknown)<br>17.28% (603,951) | Postfix<br>17.61% (618,519) | Postfix<br>1.30% (7,376) | Postfix<br>7.91% (18,284) |
| 4. | Idea<br>4.96% (170,099) | Idea<br>4.85% (169,679) | Idea<br>4.83% (169,545) | Microsoft<br>0.26% (1,443) | Microsoft<br>3.63% (8,377) |
| 5. | Microsoft<br>4.55% (156,046) | Microsoft<br>1.63% (56,930) | MailEnable<br>0.54% (18,978) | Sendmail<br>0.24% (1,363) | MailEnable<br>0.43% (999) |
| 6. | Sendmail<br>1.79% (61,188) | Sendmail<br>1.49% (51,981) | Sendmail<br>0.42% (14,888) | MailEnable<br>0.08% (438) | Sendmail<br>0.36% (832) |
| 7. | Sendinblue<br>1.71% (58,628) | Host Europe<br>0.41% (14,187) | Host Europe<br>0.40% (14,180) | (Gateway)<br>0.02% (139) | MDaemon<br>0.05% (107) |
| 8. | Sophos SMTP<br>0.84% (28,726) | MailEnable<br>0.22% (7,765) | (no banner)<br>0.19% (6,806) | MDaemon<br>0.01% (58) | OmniTI Ecelerity<br>0.02% (37) |
| 9. | MailEnable<br>0.60% (20,545) | MDaemon<br>0.13% (4,682) | MDaemon<br>0.14% (4,860) | OpenSMTPD<br>0.00% (4) | (Gateway)<br>0.01% (22) |

| Pos. | IMAP (143) | IMAPS (993) | POP3 (110) | POP3 (995) | FTP (21) | FTPS (990) |
|---|---|---|---|---|---|---|
| 1. | Dovecot<br>86.60% (3,210,657) | Dovecot<br>83.11% (3,258,031) | Dovecot<br>88.73% (3,150,958) | Dovecot<br>82.31% (3,151,001) | Pure-FTPd<br>59.91% (2,891,862) | ProFTPD<br>55.88% (170,796)<br>$<$ 1.3.5e:<br>55,28% (168,966)<br>$\geq$ 1.3.5e:<br>0.09% (271) |
| 2. | Courier<br>6.38% (236,394) | Courier<br>6.75% (264,591) | Courier<br>5.65% (200,557) | (unknown)<br>6.15% (235,525) | (unknown)<br>16.72% (806,999) | (unknown)<br>15.71% (48,009) |
| 3. | (unknown)<br>3.81% (141,183) | (unknown)<br>5.45% (213,581) | (unknown)<br>2.98% (105,776) | Courier<br>5.47% (209,594) | ProFTPD<br>15.86% (791,621)<br>$<$ 1.3.5e:<br>6.69% (333,985)<br>$\geq$ 1.3.5e:<br>0.43% (21,400) | Microsoft FTP<br>11.49% (35125) |
| 4. | Microsoft<br>0.95% (35,352) | (no banner)<br>1.26% (49,544) | Microsoft Exchange<br>0.87% (31,029) | (no banner)<br>3.19% (122,098) | Microsoft FTP<br>3.28% (158,344) | FileZilla Server<br>10.55% (32,250) |
| 5. | Cyrus<br>0.37% (13,775) | Microsoft<br>1.01% (39,643) | Cyrus<br>0.33% (11,576) | Microsoft Exchange<br>0.69% (26,337) | vsFTPd<br>1.18% (56,735) | SurgeFTP<br>2.61% (7991) |
| 6. | Kerio Connect<br>0.31% (11,661) | Cyrus<br>0.45% (17,674) | Zimbra<br>0.26% (9,196) | Mailenable<br>0.56% (21,543) | FileZilla Server<br>0.78% (37,485) | Serv-U FTP<br>2.53% (7,740) |
| 7. | Zimbra<br>0.26% (9,687) | Kerio Connect<br>0.41% (16,083) | Kerio Connect<br>0.24% (8,657) | Cyrus<br>0.34% (13,082) | Fritz!Box<br>0.44% (21,217) | vsFTPd<br>0.47% (1,425) |
| 8. | kasserver.com<br>0.23% (8,597) | Zimbra<br>0.29% (11,224) | kasserver.com<br>0.24% (8,654) | Zimbra<br>0.24% (9,044) | Serv-U FTP<br>0.42% (20,303) | Pure-FTPd<br>0.39% (1,190) |
| 9. | Training System<br>0.23% (8,467) | kasserver.com<br>0.22% (8,637) | Bigfoot<br>0.19% (6,634) | Kerio Connect<br>0.23% (8,958) | Synology<br>0.25% (12,130) | Wing FTP<br>0.34% (1,036) |

Table 7: Most popular application servers by protocol and port number from our banner scan. Only servers that complete a TLS handshake are included. For ProFTPD, we also give numbers for known exploitable and fixed versions [37] (rest is unknown).