UNIVERSITY OF CALIFORNIA, SAN DIEGO

Analyzing and addressing the security issues of non-browser web-connected applications

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Atyansh Jaiswal

Committee in charge:

Professor Deian Stefan, Chair
Professor Stefan Savage
Professor Hovav Shacham

2017

The Thesis of Atyansh Jaiswal is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

EPIGRAPH

Hardware is easy to protect: lock it in a room, chain it to a desk, or buy a spare. Information poses more of a problem. It can exist in more than one place; be transported halfway across the planet in seconds; and be stolen without your knowledge.

—Bruce Schneier

TABLE OF CONTENTS

Signature Page iii

Epigraph iv

Table of Contents v

List of Figures vii

List of Tables viii

Acknowledgements ix

Abstract of the Thesis x

Chapter 1 Saber: delegating transport layer security to browsers 1
- 1.1 Introduction 1
- 1.2 Background 3
  - 1.2.1 TLS 3
  - 1.2.2 HTTPS 4
  - 1.2.3 Strict Transport Security 5
  - 1.2.4 Public Key Pinning 6
  - 1.2.5 Certificate Revocations 7
- 1.3 Issues with non-browser web clients 8
  - 1.3.1 Certificate validation results 9
  - 1.3.2 Why do these problems exist? 12
  - 1.3.3 Non-browsers have a representation issue 13
- 1.4 Delegating connection security to browser 13
  - 1.4.1 Remote debugging protocol 14
  - 1.4.2 Linking directly to Chrome's networking code . . . 15
  - 1.4.3 Motivation 15
- 1.5 swget 16
  - 1.5.1 Secure redirects 18
  - 1.5.2 Warning messages 19
  - 1.5.3 Cookies and basic authentication 19
  - 1.5.4 Centralized HSTS and HPKP 21
  - 1.5.5 Future Work 21
- 1.6 Conclusion 22
- 1.7 Acknowledgements 22

Chapter 2 SPAM: A Secure Package Manager 23
2.1 Introduction 23
2.2 Current package managers 25
2.2.1 Malicious registry 26
2.2.2 Malicious developer 27
2.2.3 Malicious integration services 28
2.3 Registry design 28
2.3.1 The registry 28
2.3.2 User and package keys 30
2.3.3 Distributed registry 32
2.3.4 The registry threat model 33
2.3.5 Limitations 36
2.4 Related work 36
2.5 Acknowledgements 37

Bibliography 38

LIST OF FIGURES

Figure 1.1: Prototype approach 15
Figure 1.2: Optimal approach 16
Figure 1.3: Regular usage of swget 17
Figure 1.4: Behavior on certificate errors 18
Figure 1.5: Preventing unsafe https to http redirects 18
Figure 1.6: Malware detection as flagged by Chrome 19
Figure 1.7: HTTP Basic Auth 20

Figure 2.1: Example of a registry quorum 33

LIST OF TABLES

Table 1.1: Certificate error testing 11

Table 2.1: The different messages supported by our registry 30

ACKNOWLEDGEMENTS

I would like to thank my advisor and committee chair Deian Stefan for his consistent support throughout my graduate studies, and for encouraging me to pursue my thesis. I would also like to thank my other committee members Hovav Shacham and Stefan Savage for getting me interested into Computer Security as a subject and for providing valuable advise during our discussions.

I would like to thank all my co-authors Jonathan Luck, Ariana Mirian, Joshua Chao, Fraser Brown, and Andres Nötzli, for all the interesting discussions that came out of our research experience together.

I would also like to thank my friends and family for consistently providing me moral support during my time at UCSD. In particular, I'm grateful towards my parents for providing me the opportunity to move to another country for higher studies.

Chapter 1, in part is currently being prepared for submission for publication of the material. Jaiswal, Atyansh; Luck, Jonathan; Chao, Joshua; Shacham, Hovav; Stefan, Deian. The dissertation/thesis author was the primary investigator and author of this material.

Chapter 2, in part is currently being prepared for submission for publication of the material. Brown, Fraser; Mirian, Ariana; Jaiswal, Atyansh; Nötzli, Andres; Stefan, Deian. The dissertation/thesis author was a co-author of this material.

ABSTRACT OF THE THESIS

Analyzing and addressing the security issues of non-browser web-connected applications

by

Atyansh Jaiswal

Master of Science in Computer Science

University of California, San Diego, 2017

Professor Deian Stefan, Chair

Today, any non-trivial application requires the ability to communicate over the network. Providing a secure connection (i.e., a confidential and authenticated connection) for the application to achieve its goals is a difficult task as it involves correctly implementing complex protocols. Further, even if you could provide integrity and confidentiality of data received over the network, it is sometimes difficult to verify the benign nature of such data. Having stood the test of time as being the most popular application for network communication, browsers have been able to achieve network security with greater success. However, almost all other non-browser applications have lagged behind.

Despite this, these applications are widely used by developers. In this thesis, we look at two such applications.

First we look at tools that fetch webpages over https (such as wget) and analyze their connection security. We then argue that these tools should delegate network security to browsers and implement a prototype version of *wget* to demonstrate the feasibility of building applications that provides security guarantees (confidentiality, integrity, authenticity) without requiring a deep understanding of the underlying security protocols.

We then analyze package managers which developers often use to download and execute code from untrusted entities. Network security alone is not sufficient in this case. We argue for a more secure package manager, one that can cope with nation state adversaries (who have a history of infiltrating codebases). We describe the design of one such secure system—SPAM—that uses the new Stellar federated Byzantine fault tolerant system.

Chapter 1

Saber: delegating transport layer security to browsers

1.1 Introduction

Transport Layer Security (TLS) is a cryptographic protocol designed to provide end-to-end communication security between two parties, even in the presence of active man-in-the-middle adversaries [DR08]. It has since become the standard for providing secure communication on the Internet and is used for most sensitive applications, including online banking, email, instant messaging, shopping, etc. Usage of TLS is most commonly seen in securing web traffic in the form of HTTPS [Res00]. When implemented correctly, HTTPS guarantees confidentiality, integrity, and authenticity of all web communication between two parties. On the web, there are additional mechanisms such as *strict transport security* (HSTS) [HJB12], *public key pinning* (HPKP) [EPS15], *online certificate status protocol* (OCSP) stapling, and *certificate transparency* (CT) [LLK13], that improve the security of HTTPS. However implementing HTTPS and all these supporting security mechanisms correctly has proven to be a challenge for many applications.

While browsers are not free from TLS bugs, their implementations of network security protocols are regarded as the state of the art. Browser vendors have further been the pioneers of modern HTTPS standards and are the first to implement them into practice. Previous work has found that non-browser applications have struggled with validating TLS certificates [GIJ^{+}12]. As this thesis shows, many of the applications lag behind in other web security practices such as HSTS, HPKP, and certificate revocation checking. Despite this, usage of non-browser software such as package managers, command-line HTTP clients, and language specific request libraries, is prominent. These applications and libraries support communication atop TLS and hence provide the impression of giving security to the user, but they fail to live up to that promise. In contrast, we find that browsers such as Chrome and Firefox implement not only secure validation of TLS certificates in general, but also implement mechanisms such as HSTS, HPKP, verification of revoked certificates, and have dynamic protection against identified malware and dangerous binaries [PF12]. They are also better at communicating security warnings to the user in a transparent manner [FRA^{+}16].

The issues with improper implementation of TLS for non-browser applications are due to both the complexity of the protocol, lack of general understanding of TLS, and not having the same security expertise and engineering resources for development as browsers do. Solving these issues would be a monumental task and would require widespread education of the importance of properly implementing communication security. along with the expense of recruiting security experts for every application that supports HTTPS. This may not be feasible for several applications that are maintained by individual developers but are nonetheless popular.

We propose a new way to build command-line HTTP clients that allows developers to use the TLS protocol for web security but does not require the expertise needed to implement TLS and HTTPS security practices correctly. Our method involves delegating the handling of connection security to browsers, so the non-browser application deals only with application layer logic. This allows non-browser applications to get connection security “for free” from browsers. This also lets them take advantage of any mechanisms that browsers implement to harden TLS security (such as HSTS, HPKP, etc.). We have built a prototype version of wget in this fashion. Our secure wget (*swget*) application is able to validate proper TLS connections without writing any code that requires a deep knowledge of the TLS protocol, and further provide modern HTTPS security guarantees that only browsers typically provide.

This chapter is organized as follows: In Section 1.2, we discuss the background material related to TLS and HTTPS, the various existing problems with the ecosystem and how browsers have taken steps to solve them. In Section 1.3 we look at some non-browser applications, in particular *wget* to see how they compare to modern browsers in terms of connection security. In Section 1.4, we present the design of a library that delegates connection security to the browser. In Section 1.5, we present *swget* as a prototype implementation of wget that provides better connection security by delegating TLS to browsers. In Section 1.6, we conclude and discuss related work.

1.2 Background

In this section, we present a brief description TLS, HTTPS, and additional security mechanisms that harden the HTTPS ecosystem.

1.2.1 TLS

Transport Layer Security (TLS) is a cryptographic protocol that has been established as the standard method of secure, encrypted communication over the Internet [DR08]. TLS aims to provide *Authenticated Encryption* which requires the following key properties-

- **Confidentiality:** All communication between two parties is encrypted such that no information about the contents of the communication is leaked to an adversary.
- **Integrity:** An adversary must not be able to change the contents of the communication between the two parties in any way.
- **Authenticity:** The two communicating parties can verify the identity of each other (typically only the client verifies the server identity) through a certificate signed by a *trusted third-party*, and ensure that the messages exchanged are from each other. As such, an adversary cannot pretend to be one of the comunicating parties.

TLS provides these security guaranties underneath the application layer. Several application layer protocols such as HTTPS, IMAPS, XMPP, and SMTP use TLS to provide end-to-end security.

1.2.2 HTTPS

HTTPS (or "HTTP over TLS") is a protocol which secures HTTP traffic within a connection encrypted by the TLS protocol [Res00]. It is mainly used for authenticating a website and securing communication between the website and its users. This is done with the use of server certificates [CSF+08]. In a normal HTTPS connection, when a client makes a request over HTTPS to some website, the server presents the client with a certificate signed by some certificate authority. This certificate authority may not directly be trusted by the client but may instead be trusted by another entity that the client may trust. The server also responds with a chain of trust listing all the CAs that are part of the chain connecting to a root CA that the client trusts.

A client needs to validate the certificate's credentials and determine the validity of the certificate. A client may find the certificate to be invalid in cases when it's expired, has a mismatching common name field, is revoked, uses a weak signature algorithm, etc. We list several different types of client behavior for certificates in Table 1.1.

1.2.3 Strict Transport Security

While HTTPS protects users from active man-in-the-middle attackers, the protection only applies if the client actually makes an HTTPS web request. Browser extensions such as *HTTPS Everywhere* force every request to be HTTPS, but with a large portion of the web only accessible over HTTP [DKBH13], such an extension is infeasible for an average user and would block access to a significant portion of the Internet for users.

Secure connection to a domain that does serve over HTTPS can only be guaranteed if the initial web request made by the client was over HTTPS. Unencrypted HTTP requests and responses can be modified freely by an active man-in-the-middle attacker. Even if a web server redirects from HTTP to HTTPS for partial or all of the web requests that it serves, an active man-in-the-middle attacker could intercept that unencrypted redirect response and replace it with a forged accept-response pretending to be the server. As a result, the attacker could make it so that the client never actually switches to an HTTPS connection and remains vulnerable. This class of attacks is commonly known as *SSL Stripping* [Mar09].

To combat this problem, servers should also serve a *Strict-Transport-Security* (HSTS) header as part of the response that instructs the client to only visit the website over HTTPS connection [HJB12]. A client that respects this HSTS policy would automatically upgrade all HTTP requests made by the user to HTTPS before sending the request over the network. As long as the client can guarantee a secure connection with a domain once and receive an HSTS header, all future communication within a designated timeframe to that domain is protected. HSTS headers also need to provide a *max-age* policy that determines how long a client enforces the HSTS policy. This policy is continuously updated every time an HSTS header is received, so if a client visits the same website again within the expiry period, the client stays protected. Otherwise, the client loses protection when the max-age policy expires and protection is resumed once the client receives a fresh HSTS header from securely connecting to the domain again. The *max-age* policy allows websites to switch back to HTTP connections should they choose to do so in the future. Without a max-age policy, a client wouldn't be able to communicate over HTTP with the server again once it has received an HSTS header.

HSTS relies on the *Trust On First Use* (TOFU) principle to harden protection. Today, browsers actually ship with preloaded list of domains that have HSTS protection turned on by default so that users gain HSTS protection even before they establish a single secure connection. Any domain owner can apply to be a part of this list and the process has seen significant adoption since it was automated by Chrome [hst].

1.2.4 Public Key Pinning

The current HTTPS Public Key Infrastructure relies on trusting all the root CAs and by extension, all intermediate CAs trusted by root CAs. This raises the concern that if one of the CAs was coerced or compromised by some adversary, then that CA could issue a valid rogue certificate for some website without the permission of the owner of the website. This rogue certificate would be trusted by any user that trusts the CA that generated it. Previous work has shown that certificate authorities can be falsely tricked into issuing rogue certificates and in 2010, a commercial software was available for sale to government agencies to use rogue certificates for intercepting traffic [SS12]. HTTPS and HSTS alone do not protect from this kind of interception.

Instead, *HTTP Public Key Pinning* (HPKP) solves this problem by allowing web servers to serve an HTTP header over HTTPS that lists the hashes of the complete *Subject Public Key Info* field of a certificate for each public key that the domain owner wishes to use to establish a TLS connection. A client that respects the HPKP policy must ensure that while establishing a TLS connection with a website for which there is a cached HPKP entry, at least one key in the certificate chain matches a key stored in the pin set [EPS15].

This allows a domain owner to pin only their own keys. In the case that the owner chooses to only pin leaf-level keys generated by the owner themselves, no CA would be able to generate a rogue certificate for the domain that a client that has already had the keys pinned. The owner can also choose to pin the public keys for specific CAs that they trust (over all CAs), still reducing the surface area for attackers that compromise CAs.

Similar to HSTS, HPKP headers also specify a *max-age* policy which determines how long a client should remember the pinned key. This value also keeps getting continuously updated if the client receives the header again before the policy expires. HPKP headers are not very commonly seen from websites but both Firefox and Chrome provide support for it. Browsers also ship preloaded HPKP lists similar to HSTS that provide this protection by default.

1.2.5 Certificate Revocations

In the case that a domain's private key gets leaked (due to server compromise, phishing, etc.) any certificates that were issued for the corresponding public key need to be revoked. Both registering a certificate as revoked and checking the revocation status of a certificate have been tricky challenges for the server and client respectively [LTZ+15].

One way for checking revoked certificates is to maintain a Certificate Revocation List (CRL) for all revoked certificates which the clients could use to validate the certificate [CSF+08]. This list needs to be continuously updated with the latest revocations and clients need to be able to check the revocation status of a certificate dynamically. In a situation where a client is unable to access the revocation list (for example if an adversary runs a denial of service attack that prevents the client from contacting the server), then no further operations that depend upon the validation of the certificate can take place.

Online Certificate Status Protocol (OCSP) is an Internet protocol used to communicate the revocation status of x509 certificates [MAM$^+$99]. The protocol works similarly to CRLs but require less bandwidth on the client. A client requests the status of a certificate from an OCSP server, which responds whether the certificate is revoked or not. OCSP still has the same denial of service issues as CRLs and many clients tend to accept a certificate if they cannot successfully get an OCSP response. To address this issue, the original server that presents the certificate can make a request to the OCSP server and staple the OCSP response along with the original certificate, which the client can verify offline. This is known as OCSP stapling and many web servers support it today (although only about 9% of TLS connections use it as of 2015) [Goo15].

Despite OCSP stapling, an attacker that can compromise a server can simply serve the certificate without a stapled OCSP response which the client may accept. Website owners therefore have an option to issue a certificate with a "Must-Staple" flag [Gib14]. If a client sees this flag in the certificate, the client must not accept the certificate without a stapled response. This is respected by Chrome and Firefox.

Chrome and Firefox have also introduced offline CRLs that they push to user's browsers during updates [Goo15, Goo]. This is primarily used for serious incidents (say if an intermediate CA gets compromised) to push emergency updates to the CRLs maintained by the client.

1.3 Issues with non-browser web clients

TLS is a complicated protocol to implement correctly and the configuration options in TLS libraries are difficult to understand. Several non-browser softwares have been shown to be insecure against a network attacker not due to using an incorrect protocol, or a broken library, but rather due to incorrect certificate validation arising out of mistakes when configuring TLS options for their application [GIJ$^+$12].

Further, even a correct implementation of TLS does not protect these applications from SSL stripping, rogue certificates, and revocations, unless all the additional mechanisms such as HSTS, HPKP, and OCSP stapling are also implemented by the application. In our experiments with wget, curl, and the standard request libraries for Python and Node.js, we found that only wget supports HSTS and has very limited and manual support for public key-pinning. None of the four HTTP clients threw any form of error or warning for a revoked certificate and the request libraries for Python and Node.js, and curl do not have any support for HSTS and HPKP.

Wget provides support for HSTS, but upon investigation, we found that wget was not updating the expiration time for HSTS entries every time it received a valid HSTS header with a valid max-age policy. This breaks the continuity policy of HSTS which is supposed to maintain a secure HSTS entry as long as user visits the domain frequently (i.e., within the specified max-age period). If a domain provides an HSTS max-age policy of 1 day, then the wget client would be vulnerable to interception at least once a day even if the developer visits the domain multiple times within that period. We reported this issue to wget which was then promptly fixed [Ruh].

1.3.1 Certificate validation results

We tested the behavior of 6 different HTTP clients (2 command-line tools, 2 language request libraries, 2 browsers) when presented with 27 different types of certificates. The complete results are presented in Table 1.1. Below we explain different types of certificates we tested-

1. **Expired.** The certificate has an expiration date in the past.

2. **Wrong Host.** The certificate is not valid for the website that presented it and should be rejected.

3. **Self Signed.** The certificate is signed by the identity it certifies and should be rejected.

4. **Untrusted Root.** The root authority is not trusted by the client and should be rejected.

5. **Revoked.** The certificate used to be valid but was revoked by the owner and should be rejected.

6. **Pinning Test.** A different certificate has been pinned (via an HPKP header) and so this certificate should be rejected.

7. **Incomplete Chain.** The certificate chain presented is incomplete. Browsers accept this certificate since the incomplete chain was sufficient to establish trust.

8. **SHA1 Intermediate.** One of the certificates in the chain was signed using a SHA1 signing algorithm and should be rejected.

9. **1000/10000 Subject Alt Names.** The certificate provides 1000 or 10000 alternate subject names. This behavior is acceptable.

10. **RC4-MD5, RC4, 3DES, Null.** These are different types of weak cipher suites used by the respective certificates (and no cipher suite in the Null case). All of these should be rejected.

11. **DH480, DH512, DH1024, DH2048.** These are the different types of Diffie-Hellman key exchange algorithms used by the connection. Only DH2048 is currently considered completely secure.

12. **DH Small Subgroup, DH Composite.** If the key exchange was performed with a small or non-prime subgroup, then security cannot be guaranteed and these connections should be rejected.

13. **Invalid Expected SCT.** The certificate for this site uses a CA that Chrome requires valid Signed Certificate Timestamps (SCTs) for, but contains an invalid SCT. Only Chrome enforces this check.

14. **Superfish, eDellRoot, DSD Test Provider.** These are certificates issued known bad certificate authorities and should be rejected.

Table 1.1: Behavior of different clients when presented with different certificates. A green entry indicates that behavior is safe. A red entry indicates that the behavior is not ideal.

Certificate	**wget**	**curl**	**python**	**node**	**firefox**	**chrome**
Expired	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
Wrong Host	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
Self Signed	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
Untrusted Root	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
Revoked	Accepted	Accepted	Accepted	Accepted	Rejected	Rejected
Pinning Test	Accepted	Accepted	Accepted	Accepted	Rejected	Rejected
Incomplete Chain	Rejected	Accepted	Rejected	Rejected	Accepted	Accepted
SHA1 Intermediate	Accepted	Accepted	Accepted	Accepted	Rejected	Rejected
1000 Subject Alt Names	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted
10000 Subject Alt Names	Rejected	Accepted	Rejected	Rejected	Rejected	Rejected
RC4-MD5	Accepted	Rejected	Rejected	Rejected	Rejected	Rejected
RC4	Accepted	Rejected	Rejected	Rejected	Rejected	Rejected
3DES	Accepted	Accepted	Rejected	Rejected	Accepted	Accepted
Null	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
DH480	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected

Continued on next page

Table 1.1 – continued from previous page

Certificate	wget	curl	python	node	firefox	chrome
DH512	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
DH1024	Accepted	Accepted	Accepted	Accepted	Accepted	Rejected
DH2048	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted
DH Small Subgroup	Accepted	Accepted	Accepted	Accepted	Accepted	Rejected
DH Composite	Accepted	Accepted	Accepted	Accepted	Accepted	Rejected
Invalid Expected SCT	Accepted	Accepted	Accepted	Accepted	Accepted	Rejected
Superfish	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
eDellRoot	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected
DSD Test Provider	Rejected	Rejected	Rejected	Rejected	Rejected	Rejected

1.3.2 Why do these problems exist?

HSTS and HPKP are relatively recent standards that have been pioneered by web browsers so it was not surprising for us to find that non-browser software do not implement them. However, this supports the argument that browsers are ahead of other applications when it comes to providing connection security. They are also more frequently updated when compared to non-browsers allowing them to fix any discovered vulnerabilities faster.

Both Chrome and Firefox have dedicated security teams working on ensuring the safety of the respective browsers. It is difficult to for every single HTTP client implementation to afford the same attention to security as browsers can even when the applications have similar connection requirements. While we can encourage safe defaults and better tutorials, developers who implement such http clients are still prone to making mistakes when using TLS libraries. These mistakes would be difficult to eliminate since developers quite often do not have the security expertise to understand these mistakes. This can be seen by comments from developers on stack overflow, with one particular response suggesting disabling certificate validation being a top answer with hundreds of votes [sta].

1.3.3 Non-browsers have a representation issue

Due to the nature of most command-line HTTP clients and their lack of a graphical user interface, they do not convey the same information about the security of the connection to the user as browsers do. There has been a lot of work involved in presenting appropriate security warnings to the user when a certificate error occurs [FRA+16]. Further, the "green lock" symbol is also a useful indicator to users about the security of their connection to some website. All of these advantages are application specific and traditionally exclusive to browsers.

Redirects, in particular, can be easily overlooked without appropriate indicators. Browsers consider HTTPS to HTTP redirect for top-level pages as acceptable behavior since the user can look for insecure connection indicators next to the url. This behavior however may not be appropriate for an application like wget. In the case of such a redirect, wget would follow the redirect and download the resource. The redirect to HTTP is easily overlooked by an average user. Considering the original request was for an HTTPS url, the user's security may have been undermined without the knowledge of the user. If the user was downloading code and piping it to bash, a common practice, then an HTTPS to HTTP redirect could allow a man-in-the-middle attacker to run code directly on the user's machine.

1.4 Delegating connection security to browser

We propose a new way to build non-browser applications that allows developers to use the TLS protocol as implemented for web security but does not require the expertise needed to implement TLS verification correctly. We achieve this by delegating the actual network request/response handling to Chrome. Instead of directly interfacing with TLS, all requests are forwarded to Chrome, which would handle proper certificate checking on behalf of the application. This allows HTTP clients to capitalize on the security, reliability, and update frequency of Chrome, while only having to worry about the correctness of application layer logic. Another huge benefit to this approach is that the benefits are not restricted to just establishing a TLS connection. Chrome also implements HSTS, HPKP, and revocation verification, and HTTP clients delegating requests to Chrome gain all of these advantages for free as well. Chrome also provides protection from websites flagged as containing malware, phishing attacks, and dangerous binaries. These are pages that may be served over a secure connection, but are flagged as malicious (which may happen due to a server compromise, or a malicious hosting site). Delegating requests to Chrome allows other HTTP clients to avoid downloading malware as well.

1.4.1 Remote debugging protocol

We built a prototype version of *wget* that we discuss in Section 1.5. For our prototype implementation, we utilize the Chrome *remote debugging protocol*. When remote debugging is enabled for Chrome, it allows other applications to communicate with the browser over a web socket. This feature has mainly been used for debugging Chrome, but we use it to issue network requests and fetch responses. Since the network request is made by Chrome, it performs all the usual HTTPS security checks and the request only succeeds if the connection is secure, we only add additional logic for redirects. Figure 1.1 shows the design for this approach.

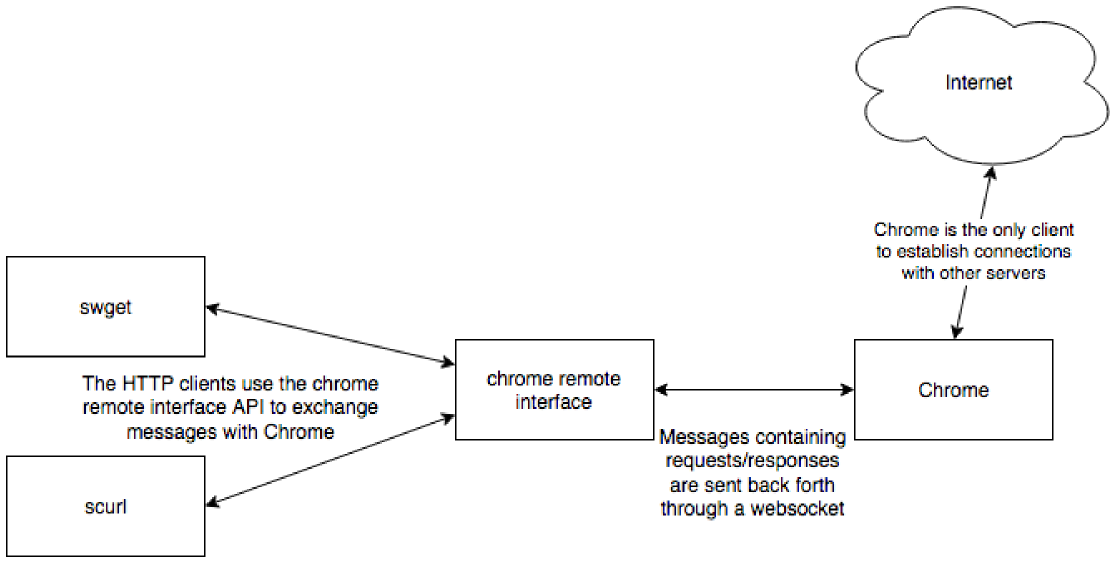

Internet
Chrome is the only client to establish connections with other servers
swget
chrome remote interface
Chrome
The HTTP clients use the chrome remote interface API to exchange messages with Chrome
Messages containing requests/responses are sent back forth through a websocket
scurl

Figure 1.1: **Prototype approach:** Applications issue requests and fetch responses from Chrome through the Chrome remote debugging protocol. Chrome makes the request on the application's behalf and fetches handles all connection security specifics.

1.4.2 Linking directly to Chrome's networking code

Using the remote debugging protocol to delegate requests to Chrome is inefficient since serialized messages are exchanged over a web socket between the HTTP client and the browser. Further, it is not perfectly secure as it allows any application to create a websocket without authentication. While this could be solved by adding an authentication mechanism to the remote debugging protocol, a more optimized and secure approach would be to link directly with Chrome's networking library. This could potentially be more lightweight as it would only require a small subset of Chrome wrapped with a thin request API that applications could interface with. Figure 1.2 shows a potential design for this approach. We leave this approach for future work.

1.4.3 Motivation

The motivation behind our delegation approach is based on two key facts: (1) Browsers are already pioneers of providing secure communication and other applications

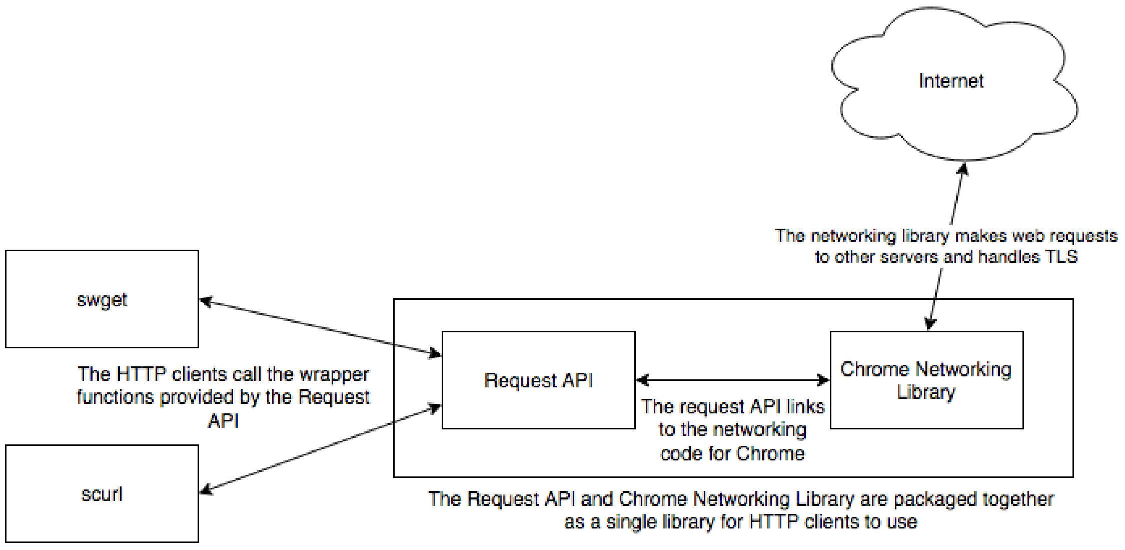
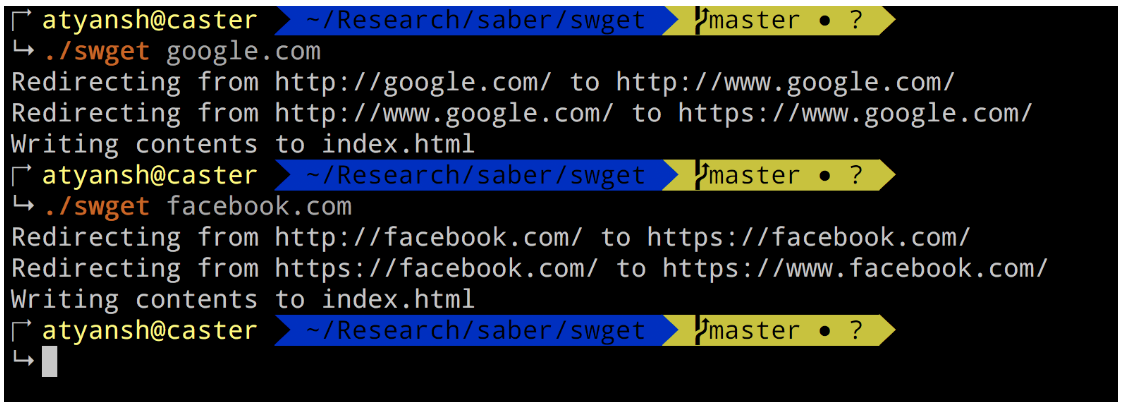

Internet
The networking library makes web requests to other servers and handles TLS
swget
The HTTP clients call the wrapper functions provided by the Request API
Request API
Chrome Networking Library
The request API links to the networking code for Chrome
scurl
The Request API and Chrome Networking Library are packaged together as a single library for HTTP clients to use

Figure 1.2: **Optimal approach:** The network request library from Chrome is exported (along with any security handling logic) and wrapped up in a library with an exposed Request API that non-browser applications interface with.

can capitalize on that. (2) Writing security critical code is difficult and we want to minimize the amount of such code that needs to be written by developers. Currently, any HTTP client would have to include its own code that either correctly implements the TLS protocol, or uses a TLS library correctly, both of which have proven to be difficult [GIJ$^+$12]. As an example of our delegation approach, we implement *swget*, a prototype of wget that provides connection security without writing any TLS verification code. This pattern can also be used to create a library that provides a simple Fetch API that any HTTP client could interface with easily.

1.5 swget

wget is a popular command line tool for downloading files from a webserver. It is frequently used by developers to crawl web pages, download scripts, and sometimes directly pipe such scripts to the shell (for example, when installing software such as Node or Bower). While such a practice of executing code obtained remotely can be dangerous in itself, it is especially a problem if the security of the connection is compromised by a man-in-the middle attacker.

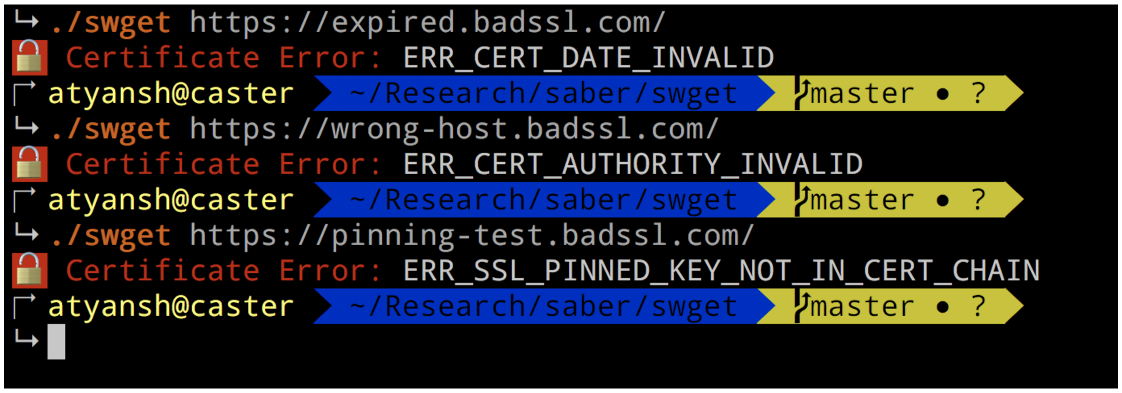


```
┌ atyansh@caster  ~/Research/saber/swget  master • ?
└→ ./swget google.com
Redirecting from http://google.com/ to http://www.google.com/
Redirecting from http://www.google.com/ to https://www.google.com/
Writing contents to index.html
┌ atyansh@caster  ~/Research/saber/swget  master • ?
└→ ./swget facebook.com
Redirecting from http://facebook.com/ to https://facebook.com/
Redirecting from https://facebook.com/ to https://www.facebook.com/
Writing contents to index.html
┌ atyansh@caster  ~/Research/saber/swget  master • ?
└→ █
```

Figure 1.3: swget fetches and saves the webpage similar to wget in the normal case

We wrote our own prototype version of *secure wget* that has the same basic functionality as wget, but uses Chrome to make network requests. Figure 1.3 shows standard behavior of swget when fetching a webpage. Looking at wget's source code, we see it has 994 lines of code are used to interface with the *OpenSSL* library. Our version of swget does not require any code to interface with a TLS library since this is done by Chrome. Only 8 lines of code are needed to add a certificate handler since Chrome would already throw an event in case any abnormal certificate is encountered. We override Chrome's certificate handling settings using the remote debugging protocol to grab the certificate error determined by Chrome and cancel the request. We used the Chrome remote interface Node.js library to construct our messages to Chrome which also contributed to the decreased code size. Figure 1.4 shows how swget responds when it catches a certificate error.

Since Chrome does certificate validation, swget's connection security is at least as restrictive as that of Chrome by default. We can manually increase or decrease restrictions based on user supplied arguments to swget. This also includes protection from revoked certificates, something which wget does not provide today. A full list of our certificate

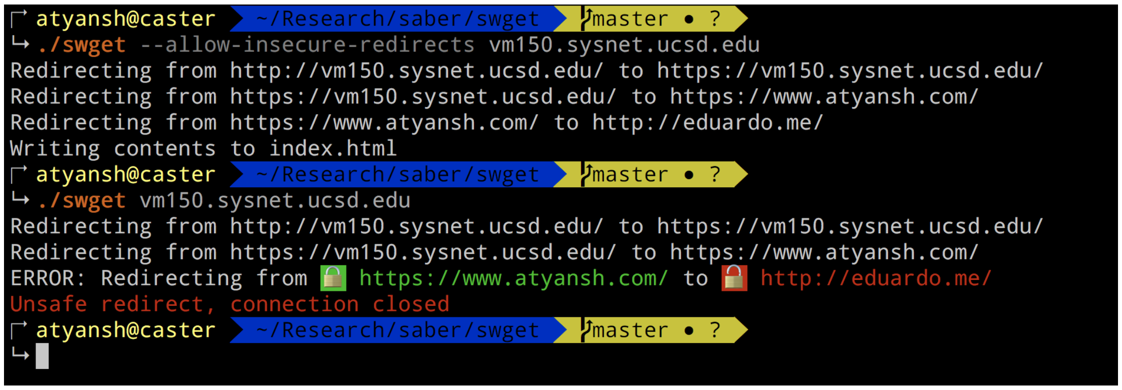


```
↳ ./swget https://expired.badssl.com/
🔒 Certificate Error: ERR_CERT_DATE_INVALID
┌ atyansh@caster  ~/Research/saber/swget  master • ?
↳ ./swget https://wrong-host.badssl.com/
🔒 Certificate Error: ERR_CERT_AUTHORITY_INVALID
┌ atyansh@caster  ~/Research/saber/swget  master • ?
↳ ./swget https://pinning-test.badssl.com/
🔒 Certificate Error: ERR_SSL_PINNED_KEY_NOT_IN_CERT_CHAIN
┌ atyansh@caster  ~/Research/saber/swget  master • ?
↳ █
```

Figure 1.4: swget catches any certificate errors caught by Chrome. In the figure, we see an example of domains with expired certificate, mismatched hostname, and an example of website that serves a valid certificate that was not pinned as part of the HPKP policy.

testing for various clients can be found in Table 1.1.

1.5.1 Secure redirects



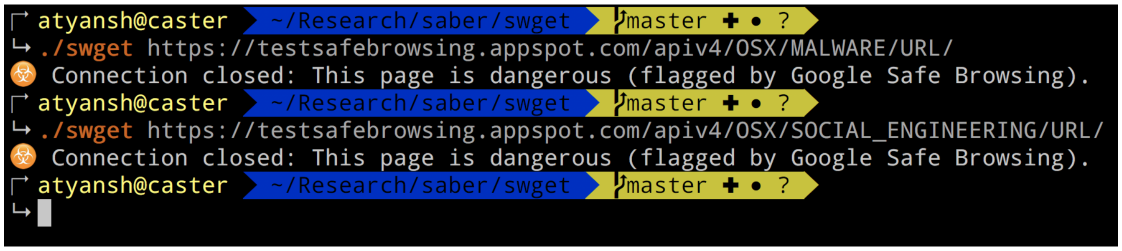
```
┌ atyansh@caster  ~/Research/saber/swget  master • ?
↳ ./swget --allow-insecure-redirects vm150.sysnet.ucsd.edu
Redirecting from http://vm150.sysnet.ucsd.edu/ to https://vm150.sysnet.ucsd.edu/
Redirecting from https://vm150.sysnet.ucsd.edu/ to https://www.atyansh.com/
Redirecting from https://www.atyansh.com/ to http://eduardo.me/
Writing contents to index.html
┌ atyansh@caster  ~/Research/saber/swget  master • ?
↳ ./swget vm150.sysnet.ucsd.edu
Redirecting from http://vm150.sysnet.ucsd.edu/ to https://vm150.sysnet.ucsd.edu/
Redirecting from https://vm150.sysnet.ucsd.edu/ to https://www.atyansh.com/
ERROR: Redirecting from 🔒 https://www.atyansh.com/ to 🔒 http://eduardo.me/
Unsafe redirect, connection closed
┌ atyansh@caster  ~/Research/saber/swget  master • ?
↳ █
```

Figure 1.5: Preventing unsafe https to http redirects

The default redirect behavior for Chrome is for top level requests is to follow redirects. While this behavior is okay for Chrome, this may not be ideal for an application like wget. Users use wget to fetch and download files. If a request to fetch was made for an https link which was then redirected to HTTP, security of the connection could be compromised by an attacker. While browsers can display the current security status to the user, there's no convenient way to check whether a file downloaded with wget was actually downloaded securely or not, once the file has already been fetched. We disable HTTPS to HTTP redirects by default in our implementation of swget. As seen from Figure 1.5 a connection is terminated if the application encounters an unsafe redirect. Command-line HTTP clients such as wget and curl do not currently provide secure redirects.

1.5.2 Warning messages

Non-browser applications have suffered from having poor UI in comparison to browsers. Previous research has shown that better browser TLS warnings decreased user click-through [FA13]. We believe that users that use non-browser applications would similarly benefit from improved warning messages and have included more visual indicators (such as lock symbols) to convey the security state of the connection. This is especially noticeable in the case of redirects (Figure 1.5) and malware warnings (Figure 1.6). Further work needs to be done to measure the effectiveness of these warnings.



```
┌ atyansh@caster  ~/Research/saber/swget  master ✚ • ?
└ ./swget https://testsafebrowsing.appspot.com/apiv4/OSX/MALWARE/URL/
☣ Connection closed: This page is dangerous (flagged by Google Safe Browsing).
┌ atyansh@caster  ~/Research/saber/swget  master ✚ • ?
└ ./swget https://testsafebrowsing.appspot.com/apiv4/OSX/SOCIAL_ENGINEERING/URL/
☣ Connection closed: This page is dangerous (flagged by Google Safe Browsing).
┌ atyansh@caster  ~/Research/saber/swget  master ✚ • ?
└ █
```

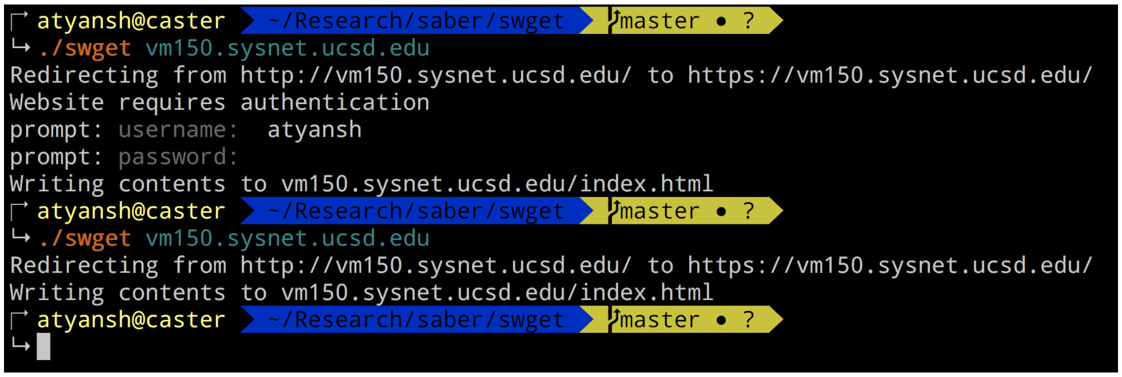
Figure 1.6: Malware detection as flagged by Chrome

1.5.3 Cookies and basic authentication

Since swget uses Chrome to issue web requests, it can take advantage of Chrome's session management. This allows us to enable/disable cookies for swget. Depending on the behavior the user wants, the user's Chrome profile can be used if the client needs to access the browser cookies, otherwise, a fresh Chrome user profile could be created that doesn't share any session state with the user's actual browser.

Cookie authentication can be pretty useful, especially to download web pages that require authentication (for example, Facebook). In standard wget, the user would have to provide their login credentials to wget or create a special authentication token for wget. In case of swget, the browser is already authenticated with the web server hence no further authentication is needed.

To demonstrate another example of authentication state management, we provide an example of HTTP basic authentication by using swget on a website that requires a user to provide login credentials before visiting the website. Using wget on this website would return a *401 Unauthorized* error unless the credentials are provided along with the url. We used a simple username/password prompt to allow an swget user to provide credentials dynamically if a website requests for them. Since Chrome remembers the authentication credentials for the session, any further requests would automatically have the credentials filled by Chrome. This can be seen in Figure 1.7. The user is prompted again if the credentials did not match correctly.



```
┌ atyansh@caster  ~/Research/saber/swget  master • ?
└ ./swget vm150.sysnet.ucsd.edu
Redirecting from http://vm150.sysnet.ucsd.edu/ to https://vm150.sysnet.ucsd.edu/
Website requires authentication
prompt: username:  atyansh
prompt: password:
Writing contents to vm150.sysnet.ucsd.edu/index.html
┌ atyansh@caster  ~/Research/saber/swget  master • ?
└ ./swget vm150.sysnet.ucsd.edu
Redirecting from http://vm150.sysnet.ucsd.edu/ to https://vm150.sysnet.ucsd.edu/
Writing contents to vm150.sysnet.ucsd.edu/index.html
┌ atyansh@caster  ~/Research/saber/swget  master • ?
└
```

Figure 1.7: HTTP Basic Auth

1.5.4 Centralized HSTS and HPKP

When Chrome receives an HSTS or HPKP header from a domain, it would record the header and provide protection for the website. Wget provides a similar HSTS protection for a domain. However, if an HSTS entry is recorded for a website by Chrome, the user gets HSTS protection only for Chrome, and not wget, and vice versa. In fact, any two different clients do not share these protections with each other. As such, the user is vulnerable every time they make an HTTP request for a website from a new client even though they have previously received an HSTS header for the domain via different client. The same problem exists for HPKP.

The model of delegating connection security to a single client—Chrome in our prototype—provides a centralized way to maintain HSTS and HPKP records. As such, any clients (such as swget) automatically share these protections with the browser. In fact, all clients that use this model would share these protections.

1.5.5 Future Work

Browsers provide protection against mixed-content in webpages (for example, HTTP hyperlinks on a webpage served over HTTPS) through security warnings. They also disable loading HTTP iframes when the top level page is served over HTTPS. Further, if a *Content-Security-Policy* header is served by the server that specifies the client to upgrade insecure requests, browsers will upgrade insecure HTTP requests to HTTPS. Swget currently does not support these mechanisms but we plan to implement CSP request upgrades in the future.

We also plan to add subresource integrity checking to swget for recursive fetching. Browsers can verify the integrity of subresources downloaded over HTTP, commonly seen for media downloaded from a CDN, by checking the cryptographic hash of the subresource obtained over HTTPS. Command-line HTTP clients can benefit greatly from subresource integrity.

1.6 Conclusion

Today, non-browser HTTP clients are weak to network adversaries in ways that browsers aren't. Our successful implementation of swget shows that this is a feasible strategy of building non-browser HTTP clients that require connection security without interfacing directly with a TLS library.

Another approach involves retrofitting implementations of SSL Libraries through dynamic linking [BPN$^+$14]. The benefits of this approach is that it doesn't require existing applications to be changed. However, this mechanism only improves proper certificate validation, it doesn't provide further protections such as malware detection, HSTS upgrade, browser maintained CRLs, etc. In contrast, our approach also address protection sharing across clients through a centralized protection mechanism.

1.7 Acknowledgements

Chapter 1, in part is currently being prepared for submission for publication of the material. Jaiswal, Atyansh; Luck, Jonathan; Chao, Joshua; Shacham, Hovav; Stefan, Deian. The dissertation/thesis author was the primary investigator and author of this material.

Chapter 2

SPAM: A Secure Package Manager

2.1 Introduction

Package managers present a security paradox: their job is to make it easy and secure to pull and install code from the internet. Uncurated package managers, package managers that do not impose restrictions on the packages that users upload, have an even tougher job precisely *because* of this lack of restriction. Uncurated managers have been wildly successful because they make it easy to install and publish packages, but they have also been subject to a number of security flaws. For example, in March of 2014, security researchers discovered a bug in npm that would allow an attacker complete control of the registry [NPM14]. This attacker, for example, could replace packages with malicious counterparts or remove packages altogether. A similar vulnerability in RubyGems came to light when a user uploaded a proof-of-concept exploit that allowed remote code execution [Eva13, pos13].

Most existing uncurated systems focus on reducing risk by securing the connection between the client and the registry to prevent network attackers from tampering with registry data in transit; npm and pip/PyPI [Pyt17, NPM16b] have both switched to making registry requests over `https` over the past few years. Similarly, most uncurated package managers check the hash of a package once it has been downloaded, a request (made over `https`) meant to ensure the integrity of the downloaded package [NPM16b]. These systems do *not* protect against registry compromise, but recent research addresses this threat. For example, the Diplomat system presents a key-based method for ensuring the integrity of packages in such an event [KTADC16].

These systems, though trying to address threats *outside* of the registry infrastructure, are not secure against internal threats: malicious or negligent maintainers and developers can compromise the registry ecosystem for all of its users. For example, in January of 2016, researchers described how evil npm software would propagate: once downloaded from the registry, this malware would add itself as a dependency to all of the infected developers' projects [NPM16c]. In a blog post, npm maintainers argue that this is a necessary side-effect of allowing arbitrary install scripts, and that "the utility of having installation scripts is greater than the risk of worms." Furthermore, "if a large number of users make a concerted effort to publish malicious packages to npm, malicious packages will be available on npm." However, embracing the dominant attitude of uncurated package managers, they also believe that "npm is largely a community of benevolent, helpful people, and so the overwhelming majority of software in the registry is safe and often useful" [NPM16c].

In contrast, other un- or lightly-curated systems have *not* found their communities to be full of benevolent, helpful people. The Google Play store struggles with data-stealing apps [Eva16], rooting apps [Goo16], and botnet apps [ML16]. Infection rates across Android app marketplaces ranged from 0.02%–0.47% in 2012, and researchers discovered multiple apps exploiting zero-day vulnerabilities during the course of this analysis [ZWZJ12]. The Chrome Extension Store faces similar problems: in 2014, Kapravelos et al. detected 130 malicious and 4,712 suspicious extensions [KGC$^+$14].

Malware and adware creators have also purchased popular Chrome extensions and then pushed automatic, malicious updates to extension users [Con14].

Similarly, registry administrators may not always act—or may be compelled *not* to act—in the best interests of their users. For example, npm administrators have re-uploaded packages after they are pulled by developers [NPM16a]. Many registries are also backed by for-profit companies[1]; trusting these registries to protect their users is like trusting Pinto-era Ford to protect its drivers—a gamble at best. Furthermore, even if npm is staffed by the opposite of Ford execs, governments may compel them to tamper with the registry anyway. At the request of the US government, for example, Yahoo! included custom backdoors to allow agencies to search user emails [Sul16]. This surveillance shows no signs of slowing (e.g., see [Tru13]).

In this chapter, we describe the current state of uncurated package managers, using npm as a running example. Then, we present designs for a community repository and package manager, SPAM, that limit the ramifications of malicious and negligent users and administrators.

2.2 Current package managers

This section presents an overview of uncurated package management systems and outlines several security challenges within these systems. We use the Node.js package manager npm as a running example because of its popularity. The security issues we describe are neither unique to npm nor specific to Node.js; most uncurated language package managers have similar drawbacks. To illustrate these common drawbacks, we first describe the workflow of a typical Node.js developer.

Developer Dan wants to reformat the columns of a CSV file. To get hip with the

[1]e.g. npm (npm, Inc.) and Yarn (Facebook)

times, he decides to use JavaScript and Node.js, and begins by writing a helper function that inserts spaces on the left-hand side of a line. The function is elegant and useful; to share it with others, he adds testing and benchmarking (pulling in dependencies like the `tape` library) and creates a new package named `lpad`.

Dan decides to make the world a better place by publishing `lpad` on the npm registry. Any developer can install Dan's `lpad` package or depend on it within their own project. Some users may even download `lpad` without knowing it; as they install more popular packages that list `lpad` as a dependency, they also install `lpad` itself. Users may even end up with multiple versions of `lpad` or multiple copies of the same version (due to how npm resolves dependencies) [npm15b, npm15a]. Eventually, fans of `lpad` start bombarding Dan with requests to update and extend his package, so he uploads it to GitHub and syncs his repository with the Travis continuous integration (CI) cloud service. Now, tests will run anytime someone creates a pull request or Dan pushes directly to the repository—`lpad` will only contain fast, functionally-correct code!

Using uncurated package managers, developers are able to build relatively complex software largely by repurposing existing projects (like Dan's `lpad`); the package manager and registry do not impose restrictions on uploaded code. This lax security, however, comes at a cost.

2.2.1 Malicious registry

In most uncurated package management systems, all users that depend on a malicious registry are at its mercy. For example, a registry may serve malicious or vulnerable packages in place of packages the users request. A registry may behave maliciously even if its administrators are virtuous: admins may be compelled by the government, network attackers may tamper with packages via MITM attacks, etc. In fact, in writing this paper, we discovered a vulnerability in npm that allows a network attacker to intercept certain installs [Ste16]; developers may explicitly link to their dependencies with `http` instead of `https` URLs, putting those who download their packages at risk. npm maintainers have not fixed this vulnerability. Instead, they warn developers to manually stick an "`s`" at the end of those links [Sel16].

2.2.2 Malicious developer

In addition to trusting the registry, developers must also trust the packages that they install, and those on which their installations depend on. This is deceptively difficult: even when developers rely on a handful of packages, their transitive dependency list is an order of magnitude larger—and that much more difficult to audit. We looked at the top 20 npm packages as of January 24, 2015, and found that the average npm package lists 50 dependencies, with a median of one, a minimum of zero, and a maximum of 626. In practice, this number is sometimes worse; for example, a popular dependency that many bootcamps include in their skeletons is `hackathon-starter` [Yal17]. This project downloads 558 dependencies before bootcampers even write a line of app code. While NPM suggests installing only trusted packages, this also does not scale when popular packages like `lodash` release every thirteen days, on average [NPM16c, SPL16]. Furthermore, packages with many dependencies may also include vulnerable and out of date dependency versions. We analyzed[2] all packages by the top ten authors on npm—presumably trustworthy people—and out of 6,498 projects, we found that 1,693, or 26.1% have out of date dependencies.

[2]The measurements and analyses described here are based on a clone of the npm registry; we fetched packages between December 6–15, 2016 using `registry-static` [dav17].

2.2.3 Malicious integration services

Along with registries and third-party code, developers also place implicit trust in integration tools like Travis [Tra17]. This implicit trust is once again transitive: for example, when Dan shares his API keys with Travis, he—and anyone whose package depends on him—is trusting Travis not to publish any malicious package versions. Furthermore, Dan must properly encrypt his keys when he shares them with Travis, since continuous integrators are allowed to use plaintext API keys. Many developers accidentally publish their API keys publicly: we searched the first 1,000 GitHub results for "provider: npm api_key." and found that 174 of 527 unique GitHub developers uploaded a YAML file with unencrypted API keys.

2.3 Registry design

In this section, we describe our secure package manager (SPAM), which intends to prevent and contain damage even in the presence of malicious users, registry administrators, and third-party tools like Travis. SPAM consists of a distributed multi-node registry which manages package data and metadata and provides an interface to update and query this data. SPAM also includes a client tool that allows developers to communicate with the registry.

2.3.1 The registry

At a high level, the goal of the registry is to maintain a ledger with information about users and packages. The ledger provides the client tools—and thus the developers who use them—with a way of verifying the authenticity of data and metadata about packages and other users. The ledger also serves to provide developers with a single mechanism for storing package metadata. We use a federated Byzantine agreement system (FBAS) to ensure that the ledger remains uncompromised in the presence of malicious actors (e.g., registry administrators or government mandates) [Maz16].

The ledger consists of different messages that record user information (e.g., name registrations and proofs of identity) and package metadata (e.g., package release information). Each entry e_n in the ledger contains the entry number n (a monotonically increasing number), a new client message m, and a hash of the entry number, message and previous ledger entry—i.e., $H(n\|m\|e_{n-1})$. Therefore, each entry in the ledger securely refers to the previous state of the ledger. The message m describes an action (e.g., update package) that the client has requested of the registry and contains that client's signature.

For example, to create a new account, a developer executes the `spam new-user` command. The SPAM client creates a new user public key pair and sends the `register_user` message to the registry; this message contains the user's public key and proposed name, signed by their corresponding private key. If that name is not already taken—if it has not already been recorded in the ledger—the registry will add the `register_user` request to the ledger. This record acknowledges that the new name is now associated with the user's key.

There are ten different messages that the SPAM client can send to the registry after interacting with users on the command line that lead to ledger modifications. Table 2.1 lists these messages, most of which are self explanatory, with the exception of the `prove_identity` and `extensible` messages—the latter of which we describe later on. The `prove_identity` message allows a user to associate their private key with their public social networks, therefore tying their SPAM identity to an external identity. For example, a developer will associate his SPAM key with his Twitter `@handle` by publishing a signed Tweet. This allows other users to verify the authenticity of the developer's packages; furthermore, it allows other users to install his packages *only if* they trust him and he has not revoked his key.

Table 2.1: The different messages supported by our registry. We omit the messages that clients use to retrieve data from the ledger.

Client message	Description
$\langle \texttt{register\_user} : U, U_{pk} \rangle_{U_{sk}}$	Register a new user U and public key U_{pk} signed with the user's private key U_{sk}.
$\langle \texttt{prove\_identity} : p, U_{pk} \rangle_{U_{sk}}$	Associate an external identity with user key U_{pk}; the proof of identity p contains URL(s) to signed social network post(s). When a registry node records this message in its ledger, it asserts to have verified the signed post(s).
$\langle \langle \texttt{register\_package} : P, P_{pk} \rangle_{P_{sk}} \rangle_{U_{sk}}$	Register a new package name P and project keys (P_{pk}, P_{sk}).
$\langle \langle \texttt{replace\_project\_key} : P, P_{pk} \rangle_{P_{sk}} \rangle_{U_{sk}}$	Revoke existing project key and associate the new key pairs (P_{pk}, P_{sk}) with project P.
$\langle \texttt{replace\_user\_key} : U, U_{pk}, p \rangle_{U_{sk}}$	Replace user U's existing key with with key-pair (U_{pk}, U_{sk}). Here, p contains proofs of identity (as above) for a majority of the U's external identities.
$\langle \langle \texttt{register\_project\_key} : P, P'_{pk} \rangle_{P'_{sk}} \rangle_{P_{sk}}$	Add a new collaborator/device key to P by registering their project key P'_{pk}. P_{sk} must correspond to the project key of the owner or another collaborator/device.
$\langle \texttt{revoke\_project\_key} : P, P'_{pk} \rangle_{P_{sk}}$	Revoke a collaborator/device key P'_{pk}. P_{sk} must correspond to the project key of the owner or another collaborator. If P'_{pk} is the owner's project key, P_{sk} must be the corresponding secret key.
$\langle \texttt{release\_package} : P, v, pkg \rangle_{P_{sk}}$	Release a new package version v for project P. The package data pkg contains the actual package tarball (and other metadata). The registry nodes record a similar message; instead of the tarball, however, they record the URL(s) where the tarball can be downloaded from.
$\langle \texttt{flag\_package} : P, v, f \rangle_{P_{sk}}$	Flag version v of package P as f (suspicious or approved), if not already flagged.
$\langle \texttt{extensible} : data \rangle_{X_{sk}}$	Add application-specific $data$ to the ledger signed by either project or user key X_{sk}.

2.3.2 User and package keys

SPAM manages all keys automatically to prevent key overuse and compromise. For example, SPAM creates a new user key when a developer creates a new account.

SPAM also generates a new key each time the user creates a new project, and *only* that key may sign project updates—the user key cannot sign them. Users can run other SPAM commands that create additional keys associated with a project, allowing them to release projects from multiple machines or integrate with CI tools like Travis. Compromise of a project key is not equivalent to compromise of a user's account; a malicious, key-stealing Travis can only push updates to a single project. Moreover, users can revoke project keys if they have been compromised.

To revoke a project key, the user runs the `spam proj revoke-key` command, which sends a message, signed with the user key, to the registry. This message indicates which project key should be revoked. At this point, the user must approve or flag any past changes to the compromised project (such as collaborator additions). In practice, this amounts to choosing an entry number to demarcate the point of compromise. The SPAM client displays a list of project changes and asks the user to flag the first suspicious change, if any. All subsequent changes are automatically flagged as suspicious. Finally, the SPAM client sends these flag messages to the registry.

To revoke and replace a user key, the user runs the `spam user replace-key` command, which consists of several steps. First, the cli tool prompts the user to update the majority of their external identity proofs with a new key (which is equivalent to sending a number of `prove_identity` messages, atomically). The cli tool will then notify the registry that the old key is invalid and that the new key is associated with the same user. By updating the ledger with the `replace_user_key` message, the registry asserts that it has verified the new key with the user's external identities (e.g., Twitter). Then, the SPAM client asks the user to demarcate a point of compromise, following similar steps to those in the project-key compromise.

2.3.3 Distributed registry

Our registry consists of several (at least four) top-level mirrors, or nodes that use a consensus protocol—specifically, the Stellar Consensus Protocol (SCP)—to agree on ledger entries [Maz16]. Like traditional, centralized Byzantine fault tolerant agreement (e.g., PBFTA [CL99]), SCP can guarantee safety and liveness by relying on a quorum of nodes to come to an agreement; for example, in an arrangement with four top-level nodes, our system tolerates one failure. Users can configure the SPAM cli tool to talk to any one of the top-level nodes, and users can trust its results as long as they configure it to receive at least two signed replies from distinct nodes.

SCP, in contrast to PBFTA, allows nodes to choose which other nodes to listen to. In combination with other design choices, this allows a node to detect and attribute attacks in which multiple registry mirrors are colluding. This is because SCP does not require a majority of nodes to compose a quorum; instead, nodes choose one of more *quorum slices* [Maz16]. Every node *n*'s quorum slices must be non-empty and contain *n* itself. For *n* to agree on ledger entry *e*, every node (including *n*) in *one* of *n*'s quorum slices must communicate that they believe *e* to be true. Now, in order for *e* to be appended to the ledger, every other node must be convinced by one of its own quorum slices. Reaching consensus is a multi-phase process, the details of which we omit; we refer the reader to the SCP paper and Stellar core for details [Maz16, ste17].

In our design, the top-level mirror nodes may employ other second-level nodes as members of other slices. Top-level mirror nodes maintain both the ledger and a mirror of the registry contents; they track package data and metadata. Second-level nodes, in contrast, do *not* store package data. Rather, they simply maintain a copy of the ledger. Second-level nodes can arbitrarily join the network (by considering at least two top level nodes to be in their quorum slice) to add additional oversight.

This additional oversight is not provided by second-level nodes referring only

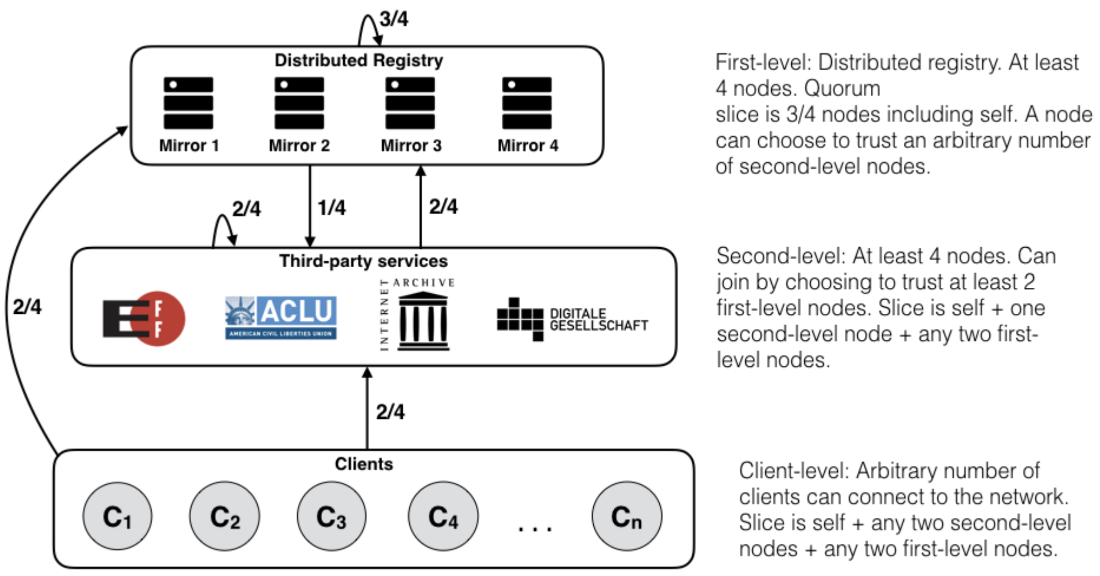

3/4
Distributed Registry
Mirror 1
Mirror 2
Mirror 3
Mirror 4
2/4
1/4
2/4
2/4
Third-party services
ARCHIVE
INTERNET
ACLU
AMERICAN CIVIL LIBERTIES UNION
DIGITALE GESELLSCHAFT
2/4
Clients
C1
C2
C3
C4
. . .
Cn
First-level: Distributed registry. At least 4 nodes. Quorum slice is 3/4 nodes including self. A node can choose to trust an arbitrary number of second-level nodes.
Second-level: At least 4 nodes. Can join by choosing to trust at least 2 first-level nodes. Slice is self + one second-level node + any two first-level nodes.
Client-level: Arbitrary number of clients can connect to the network. Slice is self + any two second-level nodes + any two first-level nodes.

Figure 2.1: Example of a registry quorum

to the ledger. Rather, all nodes also keep track of a “rollback safety” data structure that contains the previous round of signed (SCP-level) messages from nodes in its quorum slices. Since these messages contain a hash of the ledger entry histories, a signed message from a node indicates that this node agrees to the entire history of the ledger. If top-level nodes collude to rollback the ledger history, second-level nodes will be able to detect this; they will be able to prove collusion by producing two signed, contradicting messages from a culprit node. One of these messages comes from the rollback safety buffer and one is the most recent message with a backdated entry number.

2.3.4 The registry threat model

In the absence of second-level nodes, our system can only provide strong guarantees if the majority of top-level nodes are well-behaved. Under this assumption, we get availability directly from SCP’s liveness guarantees [Maz16]. SCP also ensures the safety of the ledger, which itself contains signed messages; together, this ensures the integrity and authenticity of packages. Every time a developer downloads a package, the

SPAM cli tool verifies that package's signature. Moreover, the SPAM cli tool verifies that that package has not been flagged (e.g., due to key compromise).

During the time when key is compromised but not yet revoked, our system cannot protect the developers that download malicious packages signed under the compromised key. However, SPAM's separation of project and user keys and tie in with external services—e.g., using Twitter for identity verification—make it possible to recover from key compromises without having to create a new identify. Even if an attacker steals a user key, they cannot prevent the user from revoking the key unless they compromise a majority of that user's external identities (GitHub, Twitter, StackOverflow, etc.).

In general, our system cannot provide guarantees against users running malicious code. However, we can attribute malicious packages back to their authors—and, to a certain extent, to those authors' identities. Furthermore, users can configure the SPAM cli tool to only install packages authored by developers that the user explicitly trusts—a restriction that may be imposed on sub-dependencies as well. Though this simple white-list policy is easy to ship in an early release of SPAM, we envision eventually extending the cli tool with more interesting policies. For example, we can take advantage of the fact that SPAM users are already connected to social graphs (e.g., Twitter) to compute a level of trustworthiness. Even if Eve creates fake Twitter and GitHub identities, she will have to cultivate real friendships in order to appear legitimate [3].

Without second-level nodes, a quorum of top-level mirror nodes can collude to carry out a number of attacks. For example, they can revoke and replace a user's key by appending a `replace_user_key` message to the ledger, signing it with a new, fake user-key. This is possible even without compromising the user's Twitter or GitHub accounts, because the onus of verifying identity is on the registry itself. Colluding nodes can also roll back ledger history to, for example, conceal backdoors.

[3]Here again we can leverage the idea of quorum intersection from SCP to ensure that even if Eve creates thousands of followers, if none intersect with other peoples' graphs, she will still appear suspicious.

With second-level nodes, on the other hand, our system is able to detect rollback attacks and attribute them to malicious nodes. As long as a first- or second-level node is not colluding, this node can prove that other nodes have colluded. Using the rollback safety mechanism, the honest node identifies any contradictions in any lying node's version of history. Even if all nodes except for one are colluding, honest, second-tier nodes will still see externalized messages which can be used to prove that the colluding first-level nodes in their quorum slices have said contradicting things. This alone, however, does not prevent key revocation attacks.

We can extended our system via the `extensible` message to both detect key revocation attacks and attacks wherein colluding nodes upload backdoored packages only to remove them from their registries shortly after. If third-party services (e.g., the Internet Archive, ACLU, or EFF) are willing to commit to hosting signed packages and copies of signed social network messages, they can use the `extensible` message to state their purpose in the ledger. The ACLU, for example, might say that it is willing to host this data for a certain period of time. Then, every time a package is released or an identity is proven, the ACLU will store the package or verification data on its servers and use the `extensible` message to state that it has done so. Such services enable second-level nodes to verify that the ledger's claims about key revocations and the contents of packages are replicated on the ACLU service. If registry nodes collude to revoke someone's key, the ACLU will not be able to replicate any social network data, since this data does not exist. If the SPAM cli tool does not see a such an identity proof nor a (verifiable) confirmation of revocation from ACLU, it can be configured to treat the new key as compromised: it will refuse to trust anything that the key has signed. A similar process can be used to ensure that registry nodes are not colluding to conceal backdoored packages.

2.3.5 Limitations

Our system design, naturally, comes with limitations. For example, our design does not prevent front-running attacks. These attacks occur when, for example, a developer tries to register a user name but is communicating with a malicious registry node. The malicious node will receive the user request but instead request the user's chosen name on behalf of an attacker. Once the attacker has successfully registered the name, the malicious node will handle the developer's request as usual—ensuring its denial. This problem is not an artifact of our design [SC07].

Our design also explicitly does not allow users to transfer user or project names to others (without essentially giving up on their social network identities as well). Though we believe that such extensions are possible, providing such "features" in the presence of nation-state adversaries and key continuity—i.e., the ability to easily replace compromised keys—remains an open problem.

2.4 Related work

Cappos et al. [CSBH08] identified the dangers of compromised or malicious package managers and registries. The Update Framework (TUF) [SMCD10] and the Diplomat [KTADC16] package managers—related work closest to ours—follow up on [CSBH08], borrowing ideas from Tor's update framework Thandy [Mat09]. TUF guarantees package integrity by (1) using multiple keys to sign any package data and (2) assigning roles to keys so that one key cannot be used to sign off on different kind of data (e.g., one project's key cannot be used to sign another's releases). Diplomat further extends TUF with a key hierarchy, maintained by the registry, to address an important concern—how to retrofit existing package manager design with security. We adopt the idea that user keys and project keys should be separated from these frameworks. Unlike these systems, however, we do not assume registry administrators to be trusted—our attacker model is considerably stronger. This naturally comes at the cost of not being easy to retrofit on top of existing, insecure systems.

Keybase [Key17] pioneered the idea of tying public keys with online identities. Our system uses this same idea to provide key continuity. Though we do not rely the PGP web of trust that Keybase builds on—our design is considerably simpler [Una15]—we believe that a similar idea of following or endorsing users on SPAM can potentially prove useful in assigning a level of (dis)trust to downloaded packages. For example, if a developer installs a package that is signed by a user that is not followed by anybody that the developer follows/endorses, the SPAM cli can warn before continuing with the install—potentially preventing typosquatting attacks.

2.5 Acknowledgements

Chapter 2, in part is currently being prepared for submission for publication of the material. Brown, Fraser; Mirian, Ariana; Jaiswal, Atyansh; Nötzli, Andres; Stefan, Deian. The dissertation/thesis author was a co-author of this material.

Bibliography

[BPN$^+$14] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin Butler, and Abdulrahman Alkhelaifi. Securing ssl certificate verification through dynamic linking. In *CCS*, November 2014.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, February 1999.

[Con14] Lucien Constantin. Spammers buy Chrome extensions and turn them into adware. `http://www.pcworld.com/article/2089580/spammers-buy-chrome-extensions-and-turn-them-into-adware.html`, January 2014.

[CSBH08] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. A look in the mirror: Attacks on package managers. In *CCS*. ACM, 2008.

[CSF$^+$08] 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, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[dav17] davglass. registry-static. `https://www.npmjs.com/package/registry-static`, January 2017.

[DKBH13] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the https certificate ecosystem. In *IMC*. ACM, October 2013.

[DR08] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[EPS15] C. Evans, C. Palmer, and R. Sleevi. Public key pinning extension for http. RFC 7469, RFC Editor, April 2015. `http://www.rfc-editor.org/rfc/rfc7469.txt`.

[Eva13] Evan. Data verification. `http://blog.rubygems.org/2013/01/31/data-verification.html`, January 2013.

[Eva16] Steve Evans. Data-Stealing Malicious Apps Found in Google Play Store. `https://www.infosecurity-magazine.com/news/malicious-apps-found-in-google`, September 2016.

[FA13] Adrienne Porter Felt and Devdatta Akhawe. Alice in warningland. In *USENIX*. USENIX, 2013.

[FRA$^+$16] Adrienne Porter Felt, Robert Reeder, Alex Ainslie, Helen Harris, Max Walker, Christopher Thompson, Mustafa Acer, Elisabeth Morant, and Sunny Consolvo. Rethinking connection security indicators. In *SOUPS*. USENIX, June 2016.

[Gib14] Steve Gibson. Security certificate revocation awareness: The case for ocsp must-staple. `https://www.grc.com/revocation/ocsp-must-staple.htm`, April 2014.

[GIJ$^+$12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *CCS*, October 2012.

[Goo] Google. Crlsets. `https://dev.chromium.org/Home/chromium-security/crlsets`.

[Goo15] Mark Goodwin. Revoking intermediate certificates: Introducing onecrl. `https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/`, March 2015.

[Goo16] Dan Goodin. Godless apps, some found in Google Play, can root 90Android phones. `http://arstechnica.com/security/2016/06/godless-apps-some-found-in-google-play-root-90-of-android-phones/`, June 2016.

[HJB12] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). RFC 6797, RFC Editor, November 2012. `http://www.rfc-editor.org/rfc/rfc6797.txt`.

[hst] Hsts preload. `https://hstspreload.org/`.

[Key17] Keybase. What keybase is really doing. `https://keybase.io/docs/server_security`, January 2017.

[KGC$^+$14] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security*, August 2014.

[KTADC16] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using delegations to protect community repositories. In *NSDI*, March 2016.

[LLK13] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[LTZ$^+$15] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the webs pki. In *IMC*. ACM, October 2015.

[MAM$^+$99] Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. X.509 internet public key infrastructure online certificate status protocol - ocsp. RFC 2560, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2560.txt.

[Mar09] Moxie Marlinspike. New tricks for defeating ssl in practice. https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf, 2009.

[Mat09] Nick Mathewson. Thandy: Secure update for tor. https://opensource.googleblog.com/2009/03/thandy-secure-update-for-tor.html, March 2009.

[Maz16] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, February 2016.

[ML16] Alon Menczer and Alexander Lysunets. DressCode Android Malware Discovered on Google Play. http://blog.checkpoint.com/2016/08/31/dresscode-android-malware-discovered-on-google-play, August 2016.

[NPM14] NPM. Newly paranoid maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, March 2014.

[npm15a] npm. npm v2 dependency resolution. https://docs.npmjs.com/how-npm-works/npm2, December 2015.

[npm15b] npm. npm v3 dependency resolution. https://docs.npmjs.com/how-npm-works/npm3, December 2015.

[NPM16a] NPM. kik, left-pad, and npm. http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm, March 2016.

[NPM16b] NPM. npm registry is now fully HTTPS! http://blog.npmjs.org/post/142077474335/npm-registry-is-now-fully-https, April 2016.

[NPM16c] NPM. Package install scripts vulnerability. http://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability, March 2016.

[PF12] Niels Provos and Ian Fette. All about safe browsing. https://blog.chromium.org/2012/01/all-about-safe-browsing.html, January 2012.

[pos13] postmodern. Add safe_load #119. https://github.com/ruby/psych/issues/119#issuecomment-12875715, January 2013.

[Pyt17] Python Software Foundation. PyPI - the Python package index. https://pypi.python.org/pypi, January 2017.

[Res00] E. Rescorla. Http over tls. RFC 2818, RFC Editor, May 2000. http://www.rfc-editor.org/rfc/rfc2818.txt.

[Ruh] Tim Ruhsen. Fix updating hsts entries. http://git.savannah.gnu.org/cgit/wget.git/commit/?id=57d748117ffa7bc66dedbfe8a93175a9585e2950.

[SC07] ICANN Security and Stability Advisory Committee. Advisory on domain name front running. https://www.icann.org/en/system/files/files/sac-022-en.pdf, October 2007.

[Sel16] Seldo. Avoid http urls in shrinkwrap files. http://blog.npmjs.org/post/154400916805/avoid-http-urls-in-shrinkwrap-files, December 2016.

[SMCD10] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *CCS*. ACM, October 2010.

[SPL16] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Understanding and automatically preventing injection attacks on node. js. Technical Report TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science, November 2016.

[SS12] Christopher Soghoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against ssl. In *Financial Cryptography and Data Security*. IFCA, 2012.

[sta] Trusting all certificates using httpclient over https. https://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https.

[Ste16] Deian Stefan. npm shrinkwrap allows remote execution. https://hackernoon.com/npm-shrinkwrap-allows-remote-code-execution-63e6e0a566a7#.e7an55fo2, December 2016.

[ste17] stellar. stellar-core. https://github.com/stellar/stellar-core, January 2017.

[Sul16] Margaret Sullivan. Yahoo helps the government read your emails. just following orders, they say. https://www.washingtonpost.com/lifestyle/style/yahoo-helps-the-government-read-your-emails-just-following-orders-they-say/2016/10/05/05648894-8b01-11e6-875e-2c1bfe943b66_story.html, October 2016.

[Tra17] Travis CI. Teamwork makes Travis CI possible. https://travis-ci.com/about, January 2017.

[Tru13] Donald J. Trump. https://twitter.com/realdonaldtrump/status/392990408843984897, October 2013.

[Una15] Ted Unangst. signify: Securing OpenBSD from us to you. In *BSDCan*, 2015.

[Yal17] Sahat Yalkabov. Hackathon starter - a kickstarter for nodejs web applications. https://github.com/sahat/hackathon-starter, January 2017.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, February 2012.