The Case For Secure Delegation

Dmitry Kogan, Henri Stern, Ashley Tolbert, David Mazières, and Keith Winstein
Stanford University

ABSTRACT

Today's secure stream protocols, SSH and TLS, were designed for end-to-end security and do not include a role for semi-trusted third parties. As a result, users who wish to delegate some of their authority to third parties (e.g., to run SSH clients in the cloud, or to host websites on CDNs) rely on insecure workarounds such as ssh-agent forwarding and Keyless TLS. We argue that protocol designers should consider the delegation use-case explicitly, and we propose a definition of "secure" delegation: Before a principal agrees to delegate its authority, a system should provide it with secure advance notice of **who** will do **what** to **whom** under that authority.

We developed *Guardian Agent*, a delegation system for the SSH protocol that, unlike ssh-agent forwarding, allows the user to control which delegate machines can run which commands on which servers. We were able to implement Guardian Agent in a way that remains fully compatible with existing SSH servers, by "handing over" a secure connection to the delegate once it has been set up. Additionally, we use this work to suggest a path for secure delegation on the Web.

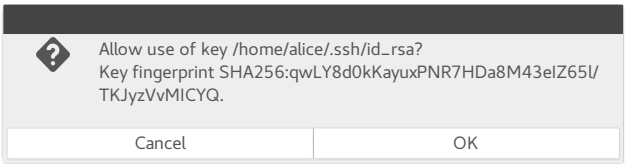
1 INTRODUCTION

Internet users often want to authorize untrusted machines to perform some operation on their behalf without giving them their credentials. For example, consider a developer who wants to pull the latest version of her private GitLab repository to her virtual machine running in Amazon's cloud. To do that, she must allow the Git client on the Amazon machine to authenticate to GitLab on her behalf. Or consider a bank that wants to serve its home page from a content delivery network (CDN) without sharing the private key of the https://www.bank.com certificate with all the edge servers.

Neither the Secure Shell protocol (SSH), used as a secure transport for Git and other services, nor the Transport Layer Security protocol (TLS), used to secure the Web, allows a user to do this securely. In our view, the fear of introducing a man-in-the-middle vulnerability has led protocol designers to create stream protocols tailored solely for secure end-to-end communication between two parties and which therefore lack first-class support for delegation.

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA

Figure 1: ssh-agent forwarding vs. Guardian Agent



(a) Current ssh-agent forwarding: when granting permission, the user doesn't know the identity of the delegate, the commands the delegate will run, or the server it will run them on.

(b) With Guardian Agent, the user has explicit control over the **who**, **what**, and **to whom** of the delegated authority, and can approve each execution individually (the **when**). The system works with existing OpenSSH servers.

The result: this void has been filled by an ecosystem of workarounds such as ssh-agent forwarding [13], shared certificates [12] and Keyless TLS [18]. Without support from the underlying protocols, these workarounds prove themselves to be insecure. For example, ssh-agent forwarding exposes the user's private keys to unauthenticated challenges without the ability to know which machine is asking to authenticate, what server it wants to authenticate to, and what command it proposes to run [4]. Workarounds that enable delegation of secure websites to CDNs are also vulnerable [6, 12].

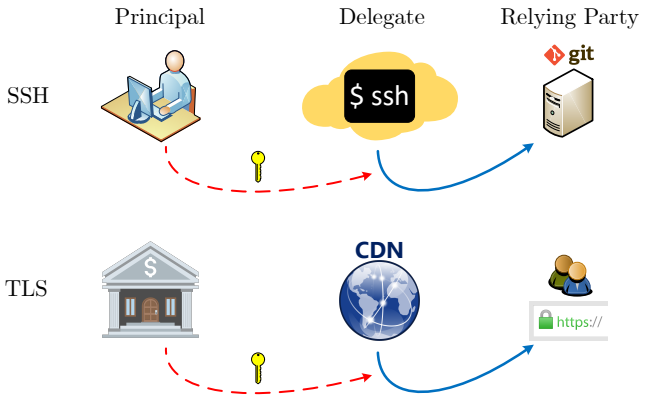
We first describe what we mean by "secure" delegation. Intuitively, most acts of delegation can be captured by a statement of the form "The principal grants the delegate permission to perform some operation on a target." A delegation mechanism should capture that intent rather than ask the user to sign a "blank check." We distill this requirement:

Secure Delegation

> Before allowing a third party to act under the principal's credentials, the principal or its agent should be able to verify and enforce: **who** is the delegate, **what** command or operation the delegate can issue **to whom**, and **when** (e.g., a single use or for a period of time).

We believe this principle ought to be uncontroversial, but while delegation is common in both the SSH and TLS settings, we are unaware of "secure delegation" mechanisms for either. In §2 we discuss existing delegation solutions for SSH and TLS in light of this principle.

Figure 2: Two common delegation scenarios



Next, we developed *Guardian Agent*, a delegation system for SSH that obeys this requirement (Figure 1). Our scheme works by first having the user's machine establish an authenticated SSH connection with the server, "locking" it to a specific command, and then handing the connection off to the untrusted delegate so it can complete the operation. Guardian Agent is open-source software[1] and is actively used at our institution. Guardian Agent installs on the user's machine and the delegate; it requires no changes to OpenSSH servers.

In summary, this paper makes the following contributions:

(1) we point to the lack of secure delegation as a problem common to both major secure stream protocols used today, SSH and TLS,

(2) we propose a definition of secure delegation, arguing that the principal should control who, under its authority, can do what to whom,

(3) we describe and evaluate Guardian Agent, a secure delegation scheme for SSH, and

(4) we present ideas on how secure delegation can be integrated into secure stream protocols more generally.

2 DELEGATION TODAY

Delegation was originally studied in the context of distributed systems [10, 17], and various formal models [1] and architecture designs [9, 21] have been proposed.

In this section, we present the current state of delegation for two common protocols (Figure 2): delegation of user credentials to untrusted SSH clients, and delegation by web servers to content distribution networks (CDNs).

2.1 SSH

Today's workflows often include work across multiple machines, trusted and untrusted. For instance, a user may want to "git pull" or "git push" from a cloud server that belongs to a third-party company. While these use-cases have become increasingly common in the Internet age, and the notion of a user's identity can no longer be tied to a physical machine, users are left without secure options to integrate secure delegation into their SSH workflows.

A naïve solution for delegation to a client machine is for the user to simply grant their credentials to it. Storing a private key on an untrusted machine allows any root user to read from disk or memory and reap the user's credentials. Even if the private key is secured by a passphrase, a root user could log the user's keystrokes or alter the binary that decrypts the key.[2] Further, copying credentials may be impossible if the user's private key is stored on a hardware security module.

Instead, the dominant system today is ssh-agent forwarding. SSH provides a program called `ssh-agent` that safeguards users' credentials on a local machine. The ssh-agent signs challenges from `ssh` processes on the local machine (allowing them to authenticate to remote servers as the user) without exposing the decrypted private key to other processes. The ssh-agent does not learn which server the process is authenticating to, or which command it intends to execute; this is acceptable because these local processes are assumed to be under the user's control.

However, SSH also provides an option (`ssh -A`) to forward ssh-agent requests from a *remote* machine (a delegate) back to a local ssh-agent. A process running on the delegate can then issue opaque signing requests to the user's ssh-agent, which signs them as if they came from a local `ssh` process.

Unfortunately, ssh-agent forwarding is not a secure delegation protocol. By enabling it, users are allowing remote machines (delegates) to authenticate as the user, without knowing which remote machine is asking, what command it will run, and what server it will run it on. A delegate that seems to be running an innocuous command (e.g., "git pull" from GitLab) could, instead, connect to a different server that the user has access to and add a rogue key to the user's authorized_keys file. The user and ssh-agent will not know the difference because the agent signs only an opaque challenge—essentially a blank check.

2.2 HTTPS Delegation to CDNs

The Transport Layer Security protocol (TLS) enables secure websites by providing an authenticated secure stream to the server and the client, with the Hypertext Transport Protocol (HTTP) usually used on top of it (HTTPS). Although TLS supports mutual authentication, it is mostly used today with one-sided server authentication, using the X.509 public key certificate infrastructure which binds domain names to public keys. When a user visits a website using an HTTPS connection, the server presents it with a certificate for the domain name requested by the user. The client then chooses a random secret key and encrypts it with the public key in the certificate. This ensures that only the legitimate owner of the domain

[1] Available at https://github.com/StanfordSNR/guardian-agent.

[2] Users can mitigate the risk of key compromise if they have cooperation from the server—for example, by creating finer-grained accounts with limited permissions, and placing keys to just those limited roles on the third-party machine. But even such mechanisms do not allow users to learn what operations are done in their names.

can decrypt the secret, thus establishing an authenticated encrypted tunnel between the browser and website.

However, current web infrastructure often has a beneficial use for a man-in-the-middle. A Content Distribution Network (CDN) is usually transparent to the browser, and is implemented by routing the browser's request to the nearest CDN edge server. The edge server serves the locally stored parts of the website directly, and proxies other requests.

What happens when a CDN needs to handle the contents of an HTTPS website? To establish an authenticated the tunnel between the browser and the CDN, the origin website needs to delegate its credentials to the CDN. Let us examine this scenario through the lens of our secure delegation principle. Our claim is that for the delegation to be secure, the origin website must be able to verify the identity of the CDN (the **who**) and the integrity of the content served by the CDN (the **what**). It should also be able to revoke the delegation (the **when**) at will, e.g., when an edge server is compromised. As CDNs usually serve the public contents of the website, the origin usually does not need to restrict the clients to which the CDN serves the data (the **to whom**), although one can also consider scenarios where client authentication is required.

Several approaches are used in this setting [12], none satisfying our notion of secure delegation:

(1) In the past, CDNs would host only HTTP content. HTTPS content would be served from the origin website. Alternately, a website may host its home page itself, while hosting less-important assets on a separate domain for which it is willing to issue a certificate to the CDN.
(2) Duplicating the private key to all the CDN edge servers. Placing the key on machines across multiple sites controlled by third-parties in multiple countries increases the risk for the website owner. The origin website has no guarantees about who is using the key and cannot verify the integrity of the contents served on its behalf.
(3) Proxying TLS handshakes. With SSL Splitting [11], the origin website performs the TLS handshake, after which it sends the encryption session key to the CDN, while keeping the data integrity key to itself [11]. It then supplies the CDN with the data integrity tags for individual data records. The downside of this scheme is that it requires using separate encryption and integrity keys and is not compatible with contemporary ciphers.
(4) In CloudFlare's Keyless TLS [18], the origin website provides the CDN with a decryption or signature oracle that the CDN uses to decrypt the session pre-master key that the browser encrypts with the public key in the certificate. Bhargavan et al. [6] describe an attack on this scheme in which an adversary that compromises a single edge server can then decrypt other sessions.

3 GUARDIAN AGENT

In this section, we present our solution to the secure delegation problem in SSH. Our system consists of a custom SSH client (which runs on the delegate) and an authentication agent (on the user's machine). Schematically, the client requests that the agent establishes an SSH connection to a server as the user, in order to run a certain command. If the user approves, the agent establishes an SSH connection to the server using its credentials, proxied at the TCP layer by the delegate, and executes the command at the client's request. The client may send or receive data directly to and from the server by asking the agent to "hand over" the encrypted connection after the command is locked in.

We use a trick to make this handoff easier: rather than defining a serialization format for the entire state of an SSH client and then transmitting that serialized state from the agent to the client, the agent simply allows the client to re-key the session.

An important property of our system is that it requires no changes to the server, and can be used as a drop-in replacement for existing SSH clients and SSH agents. This provides users with increased security without waiting for changes by SSH servers they need to interact with (e.g., GitLab).

3.1 Technical Background

The SSH protocol consists of three major components [24].

The *SSH transport layer protocol* [25] is the lowest-level component. It typically runs on top of TCP/IP, and provides higher layers with an encrypted connection that guarantees data integrity and authenticates the server to the client.

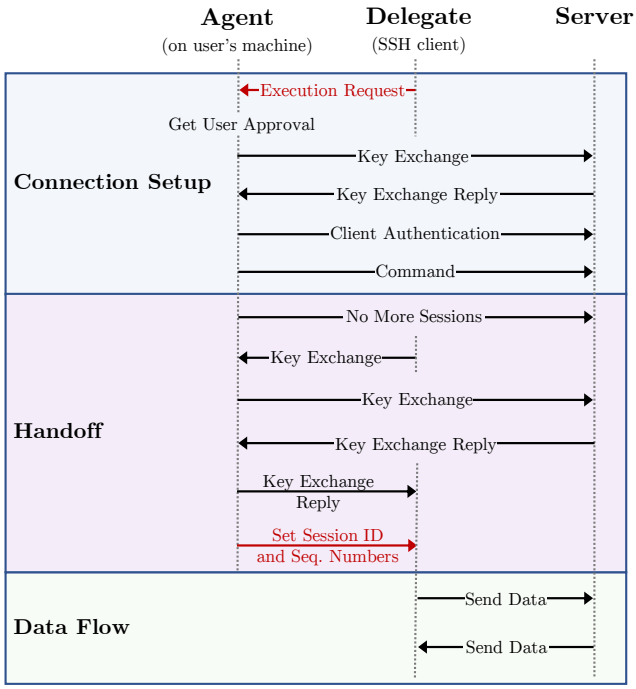
One particular feature supported by the SSH transport layer protocol is *key re-exchange*, which either party can initiate at any time during the connection by sending an `SSH_MSG_KEXINIT` packet [3]. Many SSH implementations make use of this feature periodically. The re-exchange is processed identically to the initial key-exchange, and allows changing all algorithms and keys. Furthermore, all encryption and compression contexts are reset after a key re-exchange. The session identifier remains unchanged throughout the connection. The packets of the key re-exchange itself are protected by the previously negotiated encryption and message authentication keys, until an `SSH_MSG_NEWKEYS` packet brings the new keys into effect.

The *SSH authentication protocol* [22] runs on top of the SSH transport layer protocol, and its goal is to authenticate the user to the server. Multiple authentication methods are supported, including password and public key. The authentication takes place once per connection—immediately after the initial key exchange. In particular, it is not repeated on subsequent key re-exchanges.

The *SSH connection protocol* [23] provides higher-level services such as interactive shells, execution of remote commands, forwarded TCP/IP connections and UNIX domain sockets, etc. These are implemented as independent flow-controlled channels, which the connection protocol multiplexes over the transport protocol's encrypted stream. Within the context of this protocol, a *session* is a remote execution of

[3] In the SSH context, "packet" refers to protocol messages, not IP datagrams.

Figure 3: Guardian Agent connection flow



a program, which may be a shell or other command (Figure 1 shows the command for a typical "git pull" operation).

The popular OpenSSH implementation provides a useful extension to the SSH connection protocol in the form of a `no-more-sessions` global request [13]. On receipt of this request, an OpenSSH server will refuse to open future sessions, effectively locking the connection to the commands that have already been executed. This request is rejected by non-OpenSSH servers.

3.2 System Architecture

Guardian Agent consists of a custom SSH client (running on the delegate) and an authentication agent (on the user's machine). It operates on the following three layers.

Control Layer. When the client (on the delegate machine) wishes to execute a command on some server under the user's authority, it sends a message to the agent with the username, host and port of the server and the requested command. The agent checks this request against its local security policy in order to approve or deny it.

Data Transport Layer. From the perspective of the unmodified server, the connection must take place over a single uninterrupted TCP/IP connection, whether the agent (user's machine) or client (on the delegate) is driving the show. Furthermore, the agent may not necessarily have connectivity to the server (e.g., if the server is on the same internal network as the client). The agent opens an SSH connection to the server over a TCP tunnel through the client, which acts as a TCP-level proxy. When the agent later "hands off" the SSH connection to the client, the client continues to use the same TCP connection to the server.

SSH transport and authentication. To enforce the security policy, the agent initiates the SSH connection to the server, authenticates as the user (using the credentials locally available to it), and remains in control of the connection until it is safe to hand it off to the client—which occurs after the command has been sent and "locked in." Although the connection is made through a TCP tunnel provided by the delegate, before handoff, the connection is end-to-end between the agent and server. The delegate cannot inspect or tamper with it.

3.3 Transport Handoff

In most delegation use-cases, the delegate would like to run a command on the server (e.g., "git pull" from a particular repository), and send or receive data directly. The client could, in theory, perform *all* data transmission through the agent, but this is likely to be slow if it involves substantial data transfer (§3.4). We envision the agent as running on the user's laptop with unknown connectivity; the user's goal is to authorize the delegate to speak directly to the server in a constrained way.

For this reason, after verifying the **who** (delegate), the **what** (the command) and the **to whom** (the server), and after `no-more-sessions` has been sent to prevent future command execution, the agent allows the delegate to take over the rest of the connection. We call this a transport "handoff."

When a handoff is to take place, the agent-to-server SSH connection needs to be transformed into a client-to-server connection. A naïve way to perform such a handoff would have consisted of capturing the full state of the SSH connection on the agent, transferring it to the client, and restoring it there. This state would have to include many things: the SSH session ID, the channel IDs, flow-control windows, the choice of the cryptographic algorithm suite, and the precise state of each of the ciphers and MACs used. These states are cipher-specific and are rarely exposed by current ciphers, much less in a standardized way.

This naïve handoff would have to be explicitly supported and implemented by each cipher, and would create a coupling between the cipher suite used by the client and the one used by the agent. The system would have to ensure that the agent and server end up negotiating a choice of cryptographic algorithms that is also supported by the client. This three-way compatibility would be difficult to maintain over time.

Instead, we exploit the fact that the SSH transport-layer protocol supports key re-exchange. When the client wishes to take over the connection from the agent, it sends an `SSH_MSG_KEXINIT` packet to the agent. If the agent is willing to permit a handoff, the agent forwards this packet to the server along with subsequent packets in both directions, effectively enabling a key exchange to take place *directly* between the client and the server. This key-exchange allows the client and the server to negotiate a cipher suite that is not coupled to the algorithms supported by the agent.

When key re-exchange is completed, the agent completes the handoff by sending the client a small amount of global

state that need to be synchronized explicitly: the SSH sequence number in each direction and the SSH session ID. At this point, the client can disconnect from the agent and speak directly to the server—which is none the wiser that a handoff has occurred.

The complete connection flow is illustrated in Figure 3.

3.4 Implementation

We implemented the Guardian Agent client and agent on top of the Go ssh package [2] in about 2,000 lines of Go.

The agent implements a basic policy which consists of a whitelist of allowed ⟨server, client, command⟩ entries. When a connection does not match any existing whitelist entry, the agent prompts the user on its local machine for approval. Both terminal-based and GUI prompts are provided. Figure 1 shows an example.

When used in conjunction with OpenSSH servers, which support the `no-more-sessions` feature, Guardian Agent supports a safe handoff of the connection to the client, locked to a particular command (the **what**) and one invocation of it (the **when**). When used with non-OpenSSH servers, the client can either keep the connection running through the agent for its entire duration (which carries the obvious drawback of all traffic needing to flow through the user's machine), or request a handoff without the extra security given by `no-more-sessions`. Even in this case, Guardian Agent is able to lock in the **who** and the **to whom** (the identity of the client and server), unlike current ssh-agent forwarding.

In order to provide the agent with the identity of the delegate (the **who**), we implemented a modified ssh-agent forwarding protocol, which prepends every connection to the agent with a forwarding notice that contains the identity of the remote machine from which the connection originates.[4]

4 EVALUATION

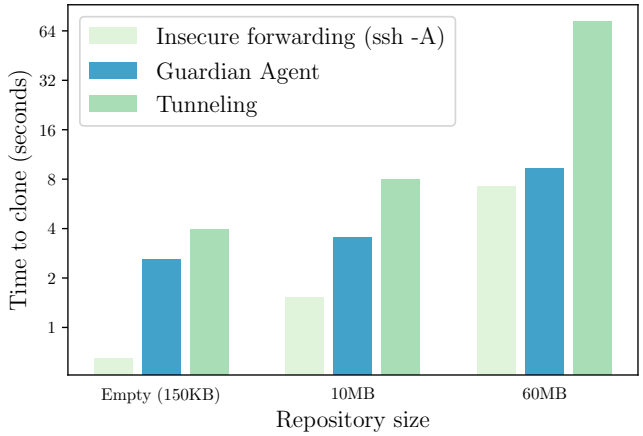
We tested Guardian Agent as a drop-in replacement for the OpenSSH `ssh` binary, both for interactive terminal sessions and for remote connections used by tools such as `git`, `scp`, `rsync`, and `mosh` [20]. To verify compatibility, we tested both our client and agent on Linux, macOS, and OpenBSD. We successfully used Guardian Agent to connect to unmodified OpenSSH servers as well as to the public SSH servers of GitHub, BitBucket, and GitLab.

Of the three commercial services, only GitLab uses an OpenSSH-based server and allows Guardian Agent to lock in the **what** (e.g., the particular repository and push vs. pull) and the **when** (limiting to one invocation). Unfortunately, GitHub and Bitbucket use alternate server implementations that do not support the `no-more-sessions` request. This means that, at present, Guardian Agent cannot restrict a delegate to push or pull to a particular repository on GitHub or Bitbucket.

To evaluate the performance cost of our protocol, we measured the time it takes to pull a Git repository to an AWS EC2 instance without disclosing the private SSH key to the EC2 instance. In the experiment, we first connected to the EC2 instance using SSH from a local laptop in our institution's network. The laptop contained the private key required to authenticate to the server. We then compared three different delegation methods (by modifying the `GIT_SSH_COMMAND` environment variable):

(1) current ssh-agent forwarding (`ssh -A`), which is fast but does not allow the agent to verify any of the facts (**who**, **what**, **to whom**, **when**) before granting approval,
(2) Guardian Agent, and
(3) tunneling all data from GitHub through the local's trusted local host, which is slow but also satisfies our principle of secure delegation.

Figure 4: Time to clone a Git repository to an untrusted delegate using three delegation methods.



We tested how the time to clone the repository scales with the repository size. The results, presented in Figure 4, show that the more complex handshake protocol of Guardian Agent increases the latency of establishing the connection. However, our results also show that Guardian Agent's performance is comparable to current forwarding, and scales significantly better than tunneling, since Guardian Agent tunnels only the initial handshake, and most data flows directly between the client and the server.

5 LIMITATIONS AND FUTURE WORK

5.1 Limitations of Guardian Agent

Guardian Agent successfully accomplishes its two main goals. First, it serves as a testbed for our secure delegation principle, and second it remedies the current state of insecure SSH delegation without having to wait for servers to adopt newer protocols. However, it has limitations, mostly from the fact that the underlying SSH protocol is left unchanged for compatibility with unmodified servers:

(1) Command enforcement (the **what** and **when**) is only possible when connecting to OpenSSH servers.
(2) Even with OpenSSH servers, the `no-more-sessions` request only prevents the creation of *sessions*, which

[4] A similar approach was suggested in a draft of the ssh-agent protocol [26].

includes command execution but not other types of channels, such as X11 and port forwarding.

(3) In order to give the agent the ability to approve or deny each command execution, our protocol cannot support session multiplexing over the same SSH connection (commonly known as the `ControlMaster` mode). As a result, a new TCP connection, handshake, and handoff are required for each command.

(4) Following an operation, a server might want to be able to prove to the principal (or to a third-party) that it received valid delegated permissions for the operation. However, this is not possible, because the SSH connection protocol is only MACed with a symmetric key, and not digitally signed.

5.2 SSH: future work

What would the SSH protocol look like if its designers had planned explicitly to allow secure delegation?

The server could explicitly accept delegated credentials, obviating the need for an agent-initiated connection and handoff. One possible design for such credentials would be special short-lived principal-generated certificates limited to a particular delegate public key (the **who**), a set of commands (the **what**), the server public key (the **to whom**) and a session-specific nonce (the **when**). The latter would be relayed to the principal by the client as part of the delegation request, and would allow the principal to restrict the credentials to a single use, without requiring the server to store additional state. Additionally, certificate chains could provide support for multi-hop delegation. This feature could be based on the "simple public-key certificate authentication system" supported by OpenSSH [14], by adding the ability to pin the public key of the server or a session-specific nonce in the certificate.

5.3 TLS delegation for HTTPS

We have discussed the limitations of existing TLS delegation schemes used by CDNs in §2.2. What would HTTPS look like if its designers had planned to allow web servers to delegate limited authority to third parties to speak on their behalf?

Secure delegation to CDNs carries the potential to reduce the amount of blind trust a publisher needs to place in the CDN, thus democratizing the CDN market by allowing new entrants to win business more quickly (and enabling secure peer-to-peer CDNs, where browsers assist other browsers).

Secure delegation for CDNs could involve an initial three-party TLS handshake between the browser, CDN, and origin (similar to Keyless TLS [18], or multi-context TLS [15]), together with a scheme that would allow the browser to verify the contents it receives from the CDN. Multiple works have focused on amending HTTP with content integrity mechanisms that would allow the browser to verify the copy of the data stored on a proxy [3, 5, 8, 16]—these could provide the **what** guarantee for downstream data. The system should also allow the server to verify the integrity of upstream data sent by the client (e.g., in POST requests). A potentially interesting protocol has been recently suggested by Eriksson et al. [7, 19]: they propose a modification to HTTP that would allow an origin server to delegate delivery of the payload of an HTTP response to a third-party.

Another question would be: to what extent can secure delegation be shoehorned into HTTPS *without* modifying browsers, and can we reuse the same tricks that worked well for SSH in this context? We hope to motivate the community's interest in such problems.

6 CONCLUSION

In this paper, we proposed a notion of "secure delegation," in which principals do not authorize third parties to take action on their behalf unless they can verify **who** will be acting under their imprimatur, **what** command will be issued **to whom**, and **when**. Although we believe this principle ought to be uncontroversial, the popular technique of ssh-agent forwarding supports none of these, and schemes like Keyless TLS enforce only the "who" (identity of the CDN).

We designed and implemented Guardian Agent, which demonstrates that it is possible to add secure delegation to SSH *without* modifying the wire protocol or the server, and without deeply instrumenting the internals of the SSH implementation (e.g., keys, nonces, other cipher state). Guardian Agent is in regular daily use by a small number of people at our organization, including the authors.

Historically, the TLS and SSH working groups have shied away from standardizing protocol features intended to encourage "man-in-the-middle" applications. The focus of both efforts has been on pure end-to-end security, but in the absence of sanctioned support for delegation use-cases, a variety of *insecure* delegation methods are now widespread in both the SSH and TLS contexts.

We believe Guardian Agent carries a hopeful message: many of the benefits of secure delegation can be gained right now, without painful changes to current protocols, as a strict security upgrade, and with modest performance impact. As a retrofit, such methods may be fragile or incomplete, and we believe our principle of who/what/to whom/when can be used to evaluate the quality of such solutions. For the future, we propose that designers should consider delegation as a first-class use-case of new secure protocols.

ACKNOWLEDGMENTS

We thank Dan Boneh, Andrew Chin, Henry Corrigan-Gibbs, John Emmons, Sadjad Fouladi, John Hood, Riad S. Wahby and Francis Yan for feedback and helpful discussions, and the HotNets reviewers for their valuable comments. This work was funded in part by the Stanford Cyber Initiative, Stanford Secure Internet of Things Project, and DARPA (grant HR0011-15-2-0047).

REFERENCES

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. 1993. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems* 15, 4 (Sep 1993), 706–734. https://doi.org/10.1145/155183.155225

[2] The Go Authors. 2011. Go ssh package. (2011). Retrieved October 16, 2017 from https://godoc.org/golang.org/x/crypto/ssh

[3] Michael Backes, Rainer W. Gerling, Sebastian Gerling, Stefan Nürnberger, Dominique Schröder, and Mark Simkin. 2014. WebTrust – A Comprehensive Authenticity and Integrity Framework for HTTP. In *Applied Cryptography and Network Security*. Springer International Publishing, 401–418. https://doi.org/10.1007/978-3-319-07536-5_24

[4] Daniel Barrett, Richard Silverman, and Robert Byrnes. 2005. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media, Inc.

[5] Roberto J. Bayardo and Jeffrey Sorensen. 2005. Merkle tree authentication of HTTP responses. In *Special interest tracks and posters of the 14th international conference on World Wide Web (WWW '05)*. 1182–1183. https://doi.org/10.1145/1062745.1062929

[6] Karthikeyan Bhargavan, Ioana Boureanu, Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. 2017. Content delivery over TLS: a cryptographic analysis of keyless SSL. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. https://doi.org/10.1109/EuroSP.2017.52

[7] Göran A.P. Eriksson, John Mattsson, Nilo Mitra, and Zaheduzzaman Sarker. 2016. Blind cache: a solution to content delivery challenges in an all-encrypted web. *Ericsson Technology Review* (Aug 2016).

[8] Camille Gaspard, Sharon Goldberg, Wassim Itani, Elisa Bertino, and Cristina Nita-Rotaru. 2009. SINE: Cache-friendly integrity for the web. In *2009 5th IEEE Workshop on Secure Network Protocols*. IEEE. https://doi.org/10.1109/npsec.2009.5342250

[9] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. 1989. The Digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*. 305–319.

[10] Morrie Gasser and Ellen McDermott. 1990. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. https://doi.org/10.1109/risp.1990.63835

[11] Chris Lesniewski-Laas and M. Frans Kaashoek. 2005. SSL splitting: Securely serving data from untrusted caches. *Computer Networks* 48, 5 (Aug 2005), 763–779. https://doi.org/10.1016/j.comnet.2005.01.006

[12] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. 2014. When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In *2014 IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/sp.2014.12

[13] Damien Miller. 2008. OpenSSH protocol vendor extensions. (Jul 2008). Retrieved October 16, 2017 from https://raw.githubusercontent.com/openssh/openssh-portable/master/PROTOCOL

[14] Damien Miller. 2010. OpenSSH certificates. (Mar 2010). Retrieved October 16, 2017 from https://raw.githubusercontent.com/openssh/openssh-portable/master/PROTOCOL.certkeys

[15] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, 199–212. https://doi.org/10.1145/2785956.2787482

[16] Kapil Singh, Helen J. Wang, Alexander Moshchuk, Collin Jackson, and Wenke Lee. 2012. Practical End-to-end Web Content Integrity. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, 659–668. https://doi.org/10.1145/2187836.2187926

[17] Karen R. Sollins. 1988. Cascaded authentication. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/secpri.1988.8108

[18] Douglas Stebila and Nick Sullivan. 2015. An Analysis of TLS Handshake Proxying. In *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 279–286. https://doi.org/10.1109/Trustcom.2015.385

[19] Martin Thomson, Göran A.P. Eriksson, and Christer Holmberg. 2016. *An architecture for secure content delegation using HTTP*. Technical Report. IETF Secretariat. https://tools.ietf.org/html/draft-thomson-http-scd-02

[20] Keith Winstein and Hari Balakrishnan. 2012. Mosh: An Interactive Remote Shell for Mobile Clients. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*. USENIX, 177–182. https://www.usenix.org/conference/atc12/technical-sessions/presentation/winstein. Available at http://mosh.org.

[21] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. 1993. Authentication in the Taos operating system. *ACM SIGOPS Operating Systems Review* 27, 5 (Dec 1993), 256–269. https://doi.org/10.1145/173668.168640

[22] Tatu Ylönen and Chris Lonvick. 2006. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. RFC Editor. http://www.rfc-editor.org/rfc/rfc4252.txt

[23] Tatu Ylönen and Chris Lonvick. 2006. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. RFC Editor. http://www.rfc-editor.org/rfc/rfc4254.txt

[24] Tatu Ylönen and Chris Lonvick. 2006. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. RFC Editor. http://www.rfc-editor.org/rfc/rfc4251.txt

[25] Tatu Ylönen and Chris Lonvick. 2006. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. RFC Editor. http://www.rfc-editor.org/rfc/rfc4253.txt

[26] Tatu Ylönen, Timo J. Rinne, and Sami Lehtinen. 2004. *Secure Shell Authentication Agent Protocol*. Internet Draft. IETF Secretariat. https://tools.ietf.org/html/draft-ietf-secsh-agent-02