TOFU for OpenPGP

Neal H. Walfield[†,♡] Werner Koch[♡]

[†] Johns Hopkins [♡] GnuPG

ABSTRACT

We present the design and implementation of a trust-on-first-use (TOFU) policy for OpenPGP. When an OpenPGP user verifies a signature, TOFU checks that the signer used the same key as in the past. If not, this is a strong indicator that a key is a forgery and either the message is also a forgery or an active man-in-the-middle attack (MitM) is or was underway. That is, TOFU can proactively detect new attacks if the user had previously verified a message from the signer. And, it can reactively detect an attack if the signer gets a message through. TOFU cannot, however, protect against sustained MitM attacks. Despite this weakness, TOFU's practical security is stronger than the Web of Trust (WoT), OpenPGP's current trust policy, for most users. The problem with the WoT is that it requires too much user support. TOFU is also better than the most popular alternative, an X.509-based PKI, which relies on central servers whose certification processes are often sloppy. In this paper, we outline how TOFU can be integrated into OpenPGP; we address a number of potential attacks against TOFU; and, we show how TOFU can work alongside the WoT. Our implementation demonstrates the practicality of the approach.

CCS Concepts

•**Security and privacy** → *Authentication; Key management;*

Keywords

Security, Authentication, OpenPGP, TOFU, MitM, Mimicry

1. INTRODUCTION

Encryption only protects against passive eavesdropping. To protect against active man-in-the-middle (MitM) attacks and forgeries, the endpoints need to authenticate each other. Currently, OpenPGP [5] users rely on the Web of Trust (WoT) to do this. In practice, the WoT offers little protection, because it requires too much user support: users need to not only exchange fingerprints, they also need to carefully designate trusted introducers. Few users take the time to do this. The most common alternative policy is an X.509-based PKI, which employs a hierarchy of trusted certificate authorities (CAs). This approach is too centralized for OpenPGP. Moreover, CAs provide weak assurances in practice.

MitM attacks are not new. However, we have observed a renewed interest in the use of OpenPGP and our (German-centric) experience is that many of the new users are interested in protecting themselves from mass surveillance, but are not technically savvy. We attribute this interest to the Snowden leaks and the CryptoParty movement. We feel that if technically sophisticated users can't be bothered to curate the WoT, then it is inappropriate for new users and a more accessible solution is not only required, but long overdue.

We propose that instead of demanding a theoretically perfect, but practically useless policy, we slightly weaken the threat model to permit less-than-perfect, but effective security policies. Concretely, we consider a threat model in which we want to protect the user from mass surveillance and opportunistic attacks, but not targeted attacks. In practice, this means that we want to protect the user from forgeries and active MitM attacks, but not sustained MitM attacks. Further, we are not interested in determining the endpoint's identity, but ensuring that the endpoint is always the same person. A trust-on-first-use (TOFU) policy satisfies these requirements and requires little user support.

In this paper, we examine how to integrate TOFU into OpenPGP. After analyzing the HCI literature, we decided that unlike `ssh`, an implementation should not ask the user to supply a policy for new bindings, but accept them automatically and only show an interception dialog when a conflict arises, which is likely to be infrequent in practice. We discuss what identifier to use for the TOFU binding (the email address included in the key) and how to prevent mimicry attacks since the attacker controls the user id. We examine how to deal with key rotation and subkeys. We discuss how to use TOFU when verifying a signature and when encrypting a message. And, finally, we explain how to combine the WoT with TOFU. We conclude that integrating TOFU into an OpenPGP implementation is not only feasible, but also usable, and hypothesize that it will better protect OpenPGP users from active MitM attacks and forgeries than the WoT of X.509-style PKI.

2. BACKGROUND

Although encryption is enough to protect users from pas-

EUROSEC'16, April 18-21 2016, London, United Kingdom

ISBN 978-1-4503-4295-7/16/04…$15.00

DOI: http://dx.doi.org/10.1145/2905760.2905761

sive adversaries, it does not prevent an active adversary from conducting a MitM attack or forging messages. For Alice to ensure that there are no eavesdroppers or impersonators, she must authenticate the other endpoint. Unfortunately, there is no mathematical process to do this. Instead, a separate, secure channel needs to be used to exchange some identity information. This requires burdensome human intervention.

The most common approach to this problem is to delegate these out-of-band checks to a few central certificate authorities (CAs). In theory, users would then authenticate and directly trust these CAs, but in practice, users just trust their user agents (Web Browser, OS, etc.) to include a list of good CAs. Using this system, users just need to confirm the connection is encrypted.

Unfortunately, there are thousands of these trusted CAs and they do a poor job. In 2011, TURKTRUST inadvertently issued two intermediate certificates instead of a regular certificate. One of these was later used for scanning encrypted traffic [8]. Between about 2010 and 2012, the Flame malware, which was developed by the USA and Israel and used for targeted attacks in the middle east, installed malicious Windows updates using a forged certificate that took advantage of a Microsoft code signing certificate that was inadvertently enabled and that still used MD5 [17]. In 2015, Symmantec, the largest issuers of certificates, admitted that it had incorrectly issued certificates for nearly a hundred domains [13]. Assuming that these problems were just due to incompetence and not malice, the most generous conclusion is that the CA system is extremely fragile and open to abuse, which we know occurs.

OpenPGP eschews these central authorizes in favor of a decentralized approach called the Web of Trust (WoT). The WoT is the social graph induced on the public, pair-wise authentication checks that users can perform. To authenticate a connection, the OpenPGP implementation just needs to find a trusted path in the WoT. When used correctly, this method can provide high confidence that a connection is secure. In practice, however, this method requires too much of a time investment for casual users: not only do these users need to verify and sign other users' fingerprints, but they also need to designate trusted introducers by setting a key's so-called *owner trust*. This burden de facto limits the utility of the WoT. But, even for users who invest the time to get this process right, the WoT's utility remains limited: it is generally unreasonable to set someone whom you have never met as a trusted introducer. Thus, at most the web of trust can be used to authenticate friends of friends.

TOFU is an authentication scheme made popular by `ssh`. Instead of confirming the identity, TOFU ensures that the key associated with an identity (a binding) does not change. The working assumption is that most connections are secure and, as such, a MitM attack or a forgery will be noticed immediately (if the binding is known) or once a good connection is established at which point damage control can be done. In other words, TOFU provides a sort of asymptotic guarantee: it cannot protect the user from sustained MitM attacks nor does it provide any guarantee about the initial connection, but the longer the user communicates with the same party, the higher the chances are that the connection is safe. TOFU also doesn't verify who a public key belongs to; it just helps ensure that communication is with the same entity.

TOFU clearly offers less theoretical protection than either an X.509-style PKI or the WoT, however, it has the huge advantage that it does not rely on third parties and requires very little user support. For the many OpenPGP users unwilling to rely on an X.509-style PKI and unwilling to invest the time to curate the WoT, TOFU would provide *significantly more protection in practice*.

3. ARCHITECTURE

3.1 The Right Amount of User Interaction

When a user uses `ssh` to connect to a host for the first time, `ssh` asks the user whether the host's key is correct. Anecdotal evidence suggests that most users accept the key without validating it. This behavior is consistent with that which that HCI researchers have come to expect: users quickly become habituated to dialogs and simply click them away [3, 2, 1]. The problem is that too many dialogs are shown in situations where the decision to ignore the warning and continue anyways has no perceived consequences [2, 1]. This is one such example: MitM attacks are rare. One could argue that some users probably do validate the host's key and thus this feature is useful to them and a minor annoyance to the rest. Unfortunately, these dialogs consume the user's attention and habituate her to taking meaningless decisions, which results in her missing important decisions [2, 1]. The general recommendation in the HCI literature is to reserve interception dialogs to high-risk activities and try particularly hard to eliminate warnings in benign situations [15, 3, 2, 1].

Based on this research, we conclude that the implementation should not ask users to provide a trust policy for new bindings, but only raise a warning when a conflict is detected. Anecdotal evidence indicates that MitM attacks and forgeries are rare, but communicating with new people is common. As such, requesting a policy for new bindings would habituate users to dismissing dialogs and eliminate any proactive protection that this scheme hopes to provide. Further, it would erode the reactive protection provided by the alarm that is raised when a conflict is detected: having habituated the user to dismissing dialogs, the user is unlikely to take the time to understand the warning, may not mark the correct key as bad, and will probably fail to do sufficient damage control.

For users who don't want positive judgments by default, it is straightforward to provide an option that configures an alternate behavior. We are aware of two reasonable alternatives. First, we can enable `ssh`'s behavior of always prompting the user for a policy when a new binding is seen. Second, we can configure the OpenPGP implementation to not assign any positive trust to bindings by default. In this case, the TOFU engine only warns about conflicts and rejects those bindings marked as being bad; it never asserts that a key is trusted. This is useful for users who actively curate the WoT and only want it to assign positive trust, but also want TOFU's ability to detect conflicts. (Section 3.6 explains how to combine these policies).

3.2 Avoiding Mimicries

In `ssh`, the user explicitly designates the host that she wants to connect to by typing in the hostname at the command prompt or selecting a previously configured connection profile. Although a host may be known by many different names (and there are no restrictions in the DNS prevent-

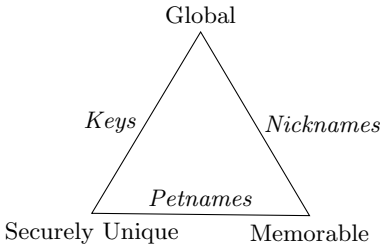

Figure 1: The petname system overlaid with Zooko's triangle [14]. Zooko's triangle describes the traits of a desirable naming system and postulates that any naming system can have at most two of these properties. Petnames, for instance, are securely unique, which makes them not susceptible to mimicry, and they are memorable, which means that the user can directly use them. But, petnames are not global; they must be assigned locally. Nicknames are global and memorable, but they are not securely unique, which means they are susceptible to mimicry attacks. Finally, keys can be made securely unique and global, but can't be directly handled by humans.

ing someone from adding more), the user typically uses the same name to designate a given host. This makes the name not only stable, but, like a *petname* [14], secure: since the name is entered by the user, it can't be altered by an attacker. These properties make this name ideal to use as the identifier in a TOFU binding.

When an OpenPGP user receives a message, she doesn't designate the sender; this is not necessary for receiving a message or verifying a signature. Instead, the identity is extracted from the OpenPGP key, which normally includes one or more valid user ids. For each of these identities, we create a TOFU binding. If any of these would create a non-benign conflict, we raise an alarm. Unfortunately, an attacker can trivially circumvent this protection. The problem is that these identifiers are just *nicknames* [14]: the key's creator controls the user id and that information is not authenticated.

Figure 1 shows the difference between petnames and nicknames.

3.2.1 Mimicry Attack

A mimicry attack is straightforward to understand. The attacker creates a key and sets the user id to something that the victim has probably not seen before. When the victim verifies a message signed with this key, the TOFU engine sees that there is a new, non-conflicting binding and assigns it the default trust policy. No warning about a possible conflict is shown, because there is no conflict.

The victim's only defense is to detect that the user id is different from what she expected. GnuPG and most mail programs that we are familiar with show at least the primary user id when verifying a message. Thus, to go undetected, the attacker needs to use an identifier that is indistinguishable from the real identifier, a so-called *mimicry* [14].

Phishing attacks have shown that creating a mimicry is easy [7, 10]. In our case, an attacker could take advantage of the fact that what humans consider to be equivalent is less strict than what a computer considers to be equivalent. For instance, many people will indicate that *John Doe* is the same identifier as *John C. Doe*. A more subtle attack is a homograph attack in which similar looking characters are substituted for expected characters [9]. For instance, most people would mistake `paypal.com` for `paypal.com` (in the first URL, the as are not Latin `a`s, but Cyrillic as).

3.2.2 Mimicry Defenses

Ideally, we wouldn't have to use the user id in the TOFU binding and could instead use a petname. The problem with petnames is that whenever the system sees a new binding, it must prompt the user to enter a petname for it. As discussed in Section 3.1, this type of interaction can result in dialog fatigue, which can undermine the security of the system.

To be able to use the user id in the TOFU binding, we need to mitigate this attack. We've come up with several methods that don't require the user to proactively supply information. First, we can force the attacker to make the mimicry more distinguishable. That is, we can make it harder to execute something like a homograph attack. Second, We can make the use of a mimicry problematic for the attacker by checking the identifier's inherent semantic meaning. Finally, we can make it more obvious when an identifier is mimicked by showing usage statistics whenever the key is used and display added warnings when the key is relatively new.

These defenses don't turn user identifiers into petnames, however, they should provide a reasonable defense against mimicry. (For those users for whom these defenses are not sufficient, TOFU is probably also not sufficient and they should instead be directly authenticating their communication partners or using the WoT.) A user study would help in determining how effective these defenses really are. We leave such a study for future work.

What's in a Name.

Before detailing our proposed defenses, we first consider what exactly we are using as an identifier. An OpenPGP key can contain one or more user ids. A user id is a free form UTF-8 string, which the OpenPGP specification recommends be in the form of an RFC 2822 mail name-addr [5], which consists of a name, an email address and a comment [11]. Most user ids are in this format. One legitimate use case for not following this recommendation is when the key is associated with a host. In this case, the user id is the host's name. Based on this, we recommend warning the user if a user id does not conform to this format. Some keys also include photo ids. We ignore these, because they are so easy to mimic (changing a bit in an image without a human noticing is trivial) and because they are typically only supplemental.

We don't want to include the comment field in the identifier. The comment field is not well defined and most users don't know how to interpret it. Although most people leave it empty, some users set it to their role (e.g., *work*), some to the key's role (e.g., *Release Signing Key*), some to their organization, and some to something random (e.g., *plantain is encrypted!*). Given this, it's unlikely that users would recognize the set of valid comments that each communication partner uses in order to identify mimicries. Moreover, newer versions of GnuPG don't ask for it when generating a key.

We also don't want to use the user's name. As previously mentioned, people consider simple variations of a name to be equivalent and there is no simple way to formalize all

of these transformations particularly given that our solution should be culturally independent. Further, names are not globally unique, which would result in false alarms.

This leaves the email address, which is actually a good identifier: it is globally unique and it has broader technical semantics, which we can exploit to help identify mimicries. Nevertheless, we still need to do some basic normalization. Although the receiver determines whether the local part of an email address is case sensitive, in practice, the local part is nearly always case insensitive. Thus, the TOFU implementation should consider John@example.org and john@example.org to be equivalent. Unfortunately, case folding is locale specific and a user id doesn't include the locale. (Encoding is generally not a problem: RFC 4880 mandates UTF-8 and only old PGP implementations didn't respect this. Thus, invalid UTF-8 strings should generate a warning.) With respect to case folding, in Turkish the lower case versions of I and $\dot{I}$ are respectively $ı$ and i (that is, I and i are *different* letters). It may be reasonable to just lowercase ASCII characters (this is what our implementation currently does), but an alternative is to use loose equivalent classes. For instance, we could consider I, $\dot{I}$, $ı$ and i equivalent even though they correspond to two distinct characters. Another simple transformation is to expand vowels with umlauts. For instance, in German *ue* is a commonly used transliteration of *ü*. These transformations result in some aliasing, however, the amount of aliasing should be limited and most such aliasing probably indicates an attack.

Making Mimicries Distinguishable.

We've identified two ways to force an attacker to use more distinguishable mimicries: compare renderings and limit character combinations.

By rendering user ids as images and comparing how similar they are, we can detect potential mimicries. Concretely, if the most similar image's similarity exceeds some threshold, then we could display a warning about a possible attack. This technique forces the attacker to use mimicries that are more distinguishable from the victim's user id in order to avoid raising an alarm. This approach requires a fair amount of work to realize and its effectiveness is unclear. We leave this as future work.

Unicode provides standard restriction levels to thwart exactly this type of attack. A restriction level defines valid character combinations. In our case, the highly restrictive level is probably best: it only allows characters from a single script or from a combination of a few select scripts [6]. (GMail has used this since 2014 [12].) This restriction level should accept most reasonable identifiers while defeating most homograph attacks. This defense would have caught the aforementioned `paypal.com` attack, which mimicked the URL by mixing Latin and Cyrillic scripts.

Checking Semantic Meaning.

An email address is more than an opaque identifier. It has semantic meaning. By checking whether this meaning is plausible and consistent with other artifacts, we can detect many mimicries.

The simplest meaning to verify is to check whether the email address is valid. If not, this is a good sign that the identifier is a mimicry. This technique has a major disadvantage: it leaks information. The amount of information leaked can be constrained by routing these verifications over something like Tor and by caching previous results.

Another approach is to have the mail client compare the identifier in in the *From* header to the email addresses listed in the key. (This technique obviously only works for emails verified within a mail client.) If none of them match, then the key is suspect. Since the *From* line can also be spoofed and replies can still be received by spoofing the *Reply-to* header, the mail client also needs to check that any address in the *Reply-to* header also appears in the key. An attacker could also spoof this line, but then he would need to make sure the spoofed email address actually works in order to receive a reply and not generate a suspicious-raising bound. This means that simple homograph attacks probably won't work, since the attacker won't have control of the domain. This forces the attacker to choose a less similar user id, which they can control, but which the user will hopefully notice as being fraudulent. Currently, we are only aware of two mail clients that implement this protection: KMail and Claws. A disadvantage of this technique is that it will raise a false alarm when checking a signature in a forwarded message.

Helping Users Recognize Mimicries.

If a mimicry doesn't cause the TOFU engine to raise an alarm, then the remaining defense is for the user to recognize the mimicry. We can support the user by displaying statistics about previous interactions with the key. If an attacker mimics the nickname of a person with whom the victim has frequent contact, then the statistics will indicate that the key is new. This should make the user suspicious.

We can collect statistics whenever the user verifies a message. Since messages may be verified multiple times (e.g., each time an email is viewed), we save the message's hash to avoid inflating the count. It is also helpful to save the time stamp embedded in the signature and the time the message was first verified. Using this information, we can compute the range of time over which messages from the key were allegedly generated and when the signatures were verified. This can reveal keys that have suddenly become active again, which may suggest a security breach.

We can also collect statistics when the user encrypts messages. In this case, saving the time the encryption was performed is sufficient. (Encrypting the same message multiple times is different from verifying a message multiple times.) These statistics can be shown in a similar way.

When checking the validity of a key, an implementation should always show the statistics, not only when the statistics are somehow suspicious. By always showing the statistics, the user gets used to seeing a steadily increasing count of verified and encrypted messages and will become surprised if this count drops to zero. Further, users become confused if security indicators aren't always shown [7].

We also recommend an extra inline warning if the number of signatures seen to be generated by the key and the number of messages encrypted to the key is less than ten. This is helpful, since successful mimicries are previously unseen keys unlike the keys they are trying to impersonate.

The statistics are also helpful when resolving conflicts. We can't assume that the new key is necessarily the bad key. If a MitM attack is broken due to a new network path, then the new key might be the legitimate one. Alternatively, if a user has received messages over many years from one key and none from another, then the new key is suspicious.

3.3 Bindings

A binding is an identity and key tuple. Associated with each binding is a policy. We have identified five useful policies: *auto*, *good*, *bad*, *unknown* and *ask*.

auto is the default policy and means that the user did not explicitly set it. We associate *marginal* trust with such bindings. The *good* policy indicates that the binding has been positively verified and, as such, is *fully* trusted. The *bad* policy is similar, but means the opposite: the key should *never* be trusted. These policies are normally only set when the user resolves a conflict (or the default policy is *ask*, see Section 3.1). (In our implementation, we also allow the user to set the policy from the command line.) The *unknown* policy is used to avoid assigning a positive trust value to a binding and corresponds to *unknown* trust. This is useful as the default policy when only the WoT should be used for making positive trust assertions, see Section 3.1. Finally, the *ask* policy means that at the next opportunity, the user should be asked to select a policy. We found this useful in our implementation when it is not possible to interact with the user (e.g., in batch mode) or when the user defers the policy decision. If the trust level is requested for a binding with this policy and it is not possible to actually ask the user, we return *undefined* trust, which means that the required information to ascertain the trust level is not available.

Subkeys.

An OpenPGP key is more than just a public key pair; it is a collection of keys and attributes. The main key is called the primary key and is used as a stable identifier. The other keys are called subkeys and are actually used for signing and encrypting. The subkeys often have specialized roles and typically have shorter lifetimes. Because the subkeys are linked to the primary key, this makes consolidation of an identity and key rotation straightforward. This property is useful for realizing a form of forward secrecy [4].

Since the identity is associated with the key as a whole rather than a particular subkey, we always use the primary key when constructing the binding even if a message is signed with a subkey or will be encrypted with a subkey. This will not interfere with normal usage patterns since the lifetime of a subkey is bounded by the lifetime of the primary key and the primary key must be available to use the subkey.

3.4 Keys

Since a key can have multiple user ids and signatures reference a key and not a user id, we need to combine the trust level of each binding to get the key's trust level. We propose taking the maximum trust level according to:

$$\textit{undefined} < \textit{unknown} < \textit{marginal} < \textit{fully} < \textit{never}$$

In this case, we ignore expired and revoked ids. If a key has no user ids, then we return *unknown* trust. However, returning the *never* trust level would also be reasonable.

Consider a concrete example: a key contains two user ids that are marked as *good* ($\Longrightarrow$ *fully* trusted). According to the above rule, the key is *fully* trusted. If the key now acquires a new user id, the new binding would be assigned the *auto* policy ($\Longrightarrow$ *marginal* trust). The trust for the key, however, remains the same. If instead, the new binding was marked as *bad*, then the whole key should be considered bad. This is exactly what this scheme decides.

3.5 Verification and Encryption

When the user verifies a signature, the implementation needs to tell the user not only if the signature is correct, but also if the key that generated the signature is trusted. We need to do the same thing when encrypting to a specified key. Although the user may designate a particular user id (rather than using the keyid), for consistency's sake, we believe it is better to compute the trust level over all bindings.

To compute the trust level for a binding, we first look to see if the binding is in the database. If so, we convert the policy to a trust level and use that. If the binding is not present, we need to check for conflicts, i.e., an existing binding with the same identifier. If there is no conflict, then we create a new binding with the default policy. Otherwise, there is a conflict and we need to ask the user what to do. Sometimes we can detect that a conflict is benign. This is the case if the user has a new key and cross signs them. Note: it is perfectly reasonable to have multiple valid keys with the same user id. This often happens when a key is rotated: normally the old key is left valid for a short time.

In the case of a non-benign conflict, the user needs to determine whether the new key is good or bad. To help the user make a more informed decision, it makes sense to display some statistics and explain the situation. At the same time, if the policy of the conflicting key was *auto*, we change its policy to *ask* and record the key that caused the conflict. The next time the trust level is retrieved for this key, the user is prompted to assign it a trust policy. One could make an argument that policies for both keys should be requested at the same time. We leave this to future work.

The above procedure assumes that the key is locally available. If this is not the case, then we can't evaluate the trust level. At most we can record information about the signature and the key and execute the above algorithm when the key becomes available. This is useful to help catch conflicts and to collect more data.

3.6 Combining the WoT and TOFU

As mentioned in Section 3.1, the WoT and TOFU can be usefully combined. Taking the maximum trust level using the ordering in Section 3.4, but inserting *expired* and *revoked* between *fully* and *never* is straightforward and seems reasonable. More complex approaches are conceivable to better take advantage of the available information, however, it's questionable whether they offer a tangible advantage. For instance, if each trust level returns two marginals, it might be reasonable to conclude that the key is *fully* trusted (in the WoT, by default, three *marginally* trusted paths to a key make it *fully* trusted instead of *marginally* trusted).

3.7 Exporting the Bindings

TOFU's strength is increased the more messages it sees over different network paths. A simple way to increase this is to share TOFU databases among users. This effectively implements something like Perspectives [16].

Unlike Perspectives in which notaries make connections to https servers and save their certificates, in OpenPGP there is no way to make a low-cost connection to a user: to get a reply, the user has to manually generate an answer. This is a significant burden particularly if it is done frequently and by many different servers.

An alternative is for users to share their TOFU databases. Unfortunately, this will expose even more of a user's social

graph than the WoT already does. (The WoT just exposes the signatures; the TOFU database exposes all communication partners and, if the statistics are included, when each communication occurred.) Further, we need a way to decide which databases to trust: if we trusted any database, then an attacker could create a fake key and upload a database that would enable him to execute a MitM attack. If we require users to explicitly select users to trust, we end up in the same situation as we are now in with the WoT: few people will bother to configure this.

4. IMPLEMENTATION

We have implemented a TOFU trust policy in GnuPG and it was initially part of the 2.1.10 release, which was released in December 2015. Our testing indicates that our implementation is able to successfully detect mimicry attacks.

5. CONCLUSION

TOFU offers better *practical* protection from active MitM attacks for OpenPGP users than the major alternatives, the WoT and X.509-style PKI. Concretely, whereas the WoT offers better theoretical protection than TOFU, in practice it requires too much user support. The result is that most users have no protection from MitM attacks most of the time. X.509, the most widespread system, simply isn't appropriate for OpenPGP, which strives as much as possible to be decentralized. Moreover, the practical security of the system leaves much to be desired. Finally, TOFU can be combined with the WoT in a straightforward manner thereby enabling the WoTs superior authentication checks when the required data is available and exploiting TOFU's consistency checks when not.

Adding TOFU support to OpenPGP presented a number of challenges. In particular, an attacker is able to control the identity used in the key-identity binding. We developed several counter-measures to mitigate this attack vector. Another difficulty is that a key may contain multiple user ids. We showed how to combine the policies from these bindings. We also explored how much user interaction is appropriate and decided that unlike `ssh`-style TOFU, the user should not be prompted to verify new bindings by default.

6. REFERENCES

[1] Devdatta Akhawe, Bernhard Amann, Matthias Vallentin, and Robin Sommer. Here's my cert, so trust me, maybe?: Understanding TLS errors on the web. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, pages 59–70, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[2] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proceedings of the 2011 Workshop on New Security Paradigms Workshop*, NSPW '11, pages 67–82, New York, NY, USA, 2011. ACM.

[3] Rainer Böhme and Stefan Köpsell. Trained to accept?: A field experiment on consent dialogs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2403–2406, New York, NY, USA, 2010. ACM.

[4] I. Brown, A. Back, and B. Laurie. Forward secrecy extensions for OpenPGP. Internet-Draft draft-brown-pgp-pfs-03, IETF Secretariat, October 2011. https://tools.ietf.org/html/draft-brown-pgp-pfs-03.

[5] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. Updated by RFC 5581.

[6] Mark Davis and Michel Suignard. Unicode security mechanisms. Technical Report Version 8.0, The Unicode Consortium, June 2015. http://www.unicode.org/reports/tr39/.

[7] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[8] Paul Ducklin. The TURKTRUST SSL certificate fiasco – what really happened, and what happens next? https://nakedsecurity.sophos.com/2013/01/08/the-turktrust-ssl-certificate-fiasco-what-happened-and-what-happens-next/, January 2013. [Online; accessed 23-March-2016].

[9] Evgeniy Gabrilovich and Alex Gontmakher. The homograph attack. *Communications of the ACM*, 45(2):128, February 2002.

[10] Zulfikar Ramzan. Phishing attacks and countermeasures. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, pages 433–448. Springer Berlin Heidelberg, 2010.

[11] P. Resnick. Internet Message Format. RFC 2822 (Proposed Standard), April 2001. Obsoleted by RFC 5322, updated by RFCs 5335, 5336.

[12] Mark Risher. Protecting Gmail in a global world. http://googleforwork.blogspot.de/2014/08/protecting-gmail-in-global-world.html, August 2014. [Online; accessed 23-March-2016].

[13] Ryan Sleevi. Sustaining digital certificate security. https://security.googleblog.com/2015/10/sustaining-digital-certificate-security.html, October 2015. [Online; accessed 23-March-2016].

[14] Marc Stiegler. An introduction to petname systems. http://www.skyhunter.com/marcs/petnames/IntroPetNames.html, February 2005 (updated June 2010).

[15] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of SSL warning effectiveness. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 399–416, Berkeley, CA, USA, 2009. USENIX Association.

[16] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.

[17] Wikipedia. Flame (malware) — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 23-March-2016].