

The Case for Transient Authentication

Brian D. Noble and Mark D. Corner
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI
{bnoble,mcorner}@umich.edu

Abstract

How does a machine know who is using it? Currently, systems assume that the user typing now is the same person who supplied a password days ago. Such persistent authentication is inappropriate for mobile and ubiquitous systems, because associations between people and devices are fleeting. To address this, we propose transient authentication. In this model, a user wears a small hardware token that authenticates the user to other devices over a short-range, wireless link. This paper presents the four principles of transient authentication, our experience applying the model to a cryptographic file system, and our plans for extending the model to other services and applications.

1 Introduction

How does a device know that the person using it is the right person? Unfortunately, authentication between people and their devices is both *infrequent* and *persistent*. Should a device fall into the wrong hands, the imposter has the full rights of the legitimate user while authentication holds.

To see why, first consider how two computational principals authenticate one another's messages. Each principal knows some shared secret, a *session key*. The sender uses this key to compute a *message authentication code*, or MAC, which is embedded in each sent message [11]. The receiver recomputes the MAC to verify that the presumed sender did in fact send that particular message. Each message is inseparably bound with the proof of its authenticity; authentication is *atomic*.

Unfortunately, it is infeasible to ask users to provide authentication for each request made of a device. Imagine a system that required the user to manually compute a MAC for each command. Instead, users authenticate *infrequently* to devices. User authentication holds until it is explicitly revoked, though some systems further limit its duration to hours or days—it is *persistent*.

Persistent authentication has been acceptable for personal computing because PCs have relatively strong physical security. For example, it is likely that the person typing at the keyboard of my office workstation is someone that I trust. However, mobile devices are easily carried, and therefore easily lost or stolen; if someone steals your laptop while you are logged in, they have full access to your data. Likewise, ubiquitous computing elements are often public, accessible to trusted and untrusted users alike.

One way to limit the vulnerabilities of persistent authentication is to limit its duration. This increases the user's burden, encouraging him to disable security entirely. By default, Windows 2000 asks users to reauthenticate whenever a laptop awakens from suspension. Anecdotally, we find that many people disable this feature, forfeiting its protection for ease of use.

Persistent authentication creates tension between protection and usability. To maximize protection, a device must constantly reauthenticate its user. To be usable, authentication must be long-lived. We resolve this tension with a new model, called *transient authentication*.

In this model of authentication, a user wears a small token, such as the IBM Linux watch [9], equipped with a short-range wireless link and modest computational resources. This token is able to authenticate constantly on the user's behalf. It also acts as a proximity cue to applications and services; if the token does not respond to an authentication request, the device can take steps to secure itself.

At first glance, transient authentication merely seems to shift the problem of authentication to the token. However, mobile and ubiquitous devices are not physically bound to any particular user; either they are carried or they are part of the surrounding infrastructure. As long as the token can be unobtrusively worn, it affords a greater degree of physical security.

We first enumerate the principles underlying transient authentication. We then briefly describe a proof-of-concept cryptographic file system that uses this authentication model, and our plans for developing an API to support a broad range of services and applications.

2 Transient Authentication Principles

Transient authentication consists of four properties. First, users must hold the sole means to access resources on the device. Second, the system must impose no additional usability burdens. Third, the mechanisms to secure and restore sensitive data on a mobile device need be no faster than the people using it. Fourth, users must give explicit consent to later actions performed on their behalf.

2.1 Tie Capabilities to Users

The ability to perform sensitive operations must reside with the user rather than his devices. For example, the keys to decrypt private data must reside on the user's token, and not on some other device. Each protected application and service must be structured in such a way as to depend on capabilities that reside on the token.

At the same time, it is unlikely that the token—a small, embedded device—can perform large computations such as bulk decryption. Furthermore, requiring the token to perform cryptographic operations in the critical path of common actions will lead to unacceptable latency. In such cases, it may be necessary to cache capabilities on a device for performance. Most often, cached capabilities are obtained through a cryptographic operation using keys only on the token. The decrypted capabilities must be destroyed when the user (and his token) leave, and the master capability should not be exposed beyond the token.

The token and device exchange capabilities using a wireless link. The system must provide confidentiality and integrity for these messages. This is ensured by a session key shared by the token and device, used to encrypt each packet, and to generate a MAC.

One could instead imagine a simple token that responded to authentication challenges. This gives evidence of the user's presence, but does not supply a cryptographic capability. An operating system could use this evidence to govern access to resources, data, and services. Unfortunately, this model is insufficient. If the device is *capable* of acting without the token, then an attacker with physical possession can potentially force it to do so. As a simple example, consider file system access control. An unencrypted disk can be removed and inspected on a machine that does not check for the token's presence. An encrypted disk, with the keys stored only on the token, is not subject to the same attack.

2.2 Do No Harm

Investing capabilities with users increases the security of the system. However, increases in security cannot impose increased user burdens. When faced with inconvenience, however small, users are quick to disable or work around

security mechanisms. Not only must the performance of the machine remain unaffected, but additional usability burdens are unacceptable.

Users already accept infrequent tasks required for security. For instance, passwords are used occasionally, usually on the order of once a day. More frequent requests for passwords are perceived as burdensome; a transparent authentication system should impose no more usability constraints than current systems.

Transient authentication must also preserve performance, despite the additional computation increased security requires. As long as this computation is imperceptible to the user, it is an acceptable burden. For example, the Secure Socket Layer (SSL) [5] protocol requires processing time for encryption and authentication. This cost is easily masked by the latency of loading web pages.

2.3 Secure and Restore on People Time

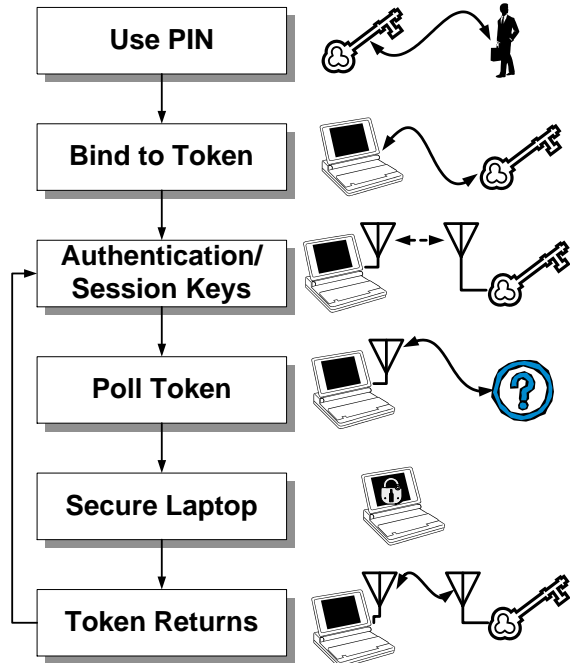
Cached capabilities—and the data they protect—can only remain while the token is present; when the token is out of range, sensitive items must be protected. This process must happen before an attacker gains access to the machine. One might think that this must happen as quickly as possible. However, since people are slow, the limit is on the order of seconds, not milliseconds.

Rather than simply erasing sensitive information, one might prefer to encrypt and retain it. This additional work can save time on restoration: when the user returns, one can obtain the proper key from the token and decrypt the data in place, restoring the machine to pre-departure state. Since the restoration process begins when the user re-enters radio range, it can complete before the user resumes work.

2.4 Ensure Explicit Consent

Tokens and devices must interact securely, and with the user's knowledge. In a wireless environment, it is particularly dangerous to carry a token that could provide capabilities to unknown devices autonomously. A "tailgating" attacker could force a user's token to provide secret keys, nullifying the security of the system. Instead, the user must authorize individual requests from devices or create trust agreements between individual devices and the token.

On one hand, users could confirm every capability requested by the device. However, usability is paramount, thus the granularity of authorization must be much larger. Instead of an action by action basis, user consent can be given on a device by device basis. If this granularity is made smaller, more usability demands would be placed on the user with no corresponding gain in security. For instance, once a sensitive key has been given to a laptop, other programs on the machine can access that key by corrupting the operating system.

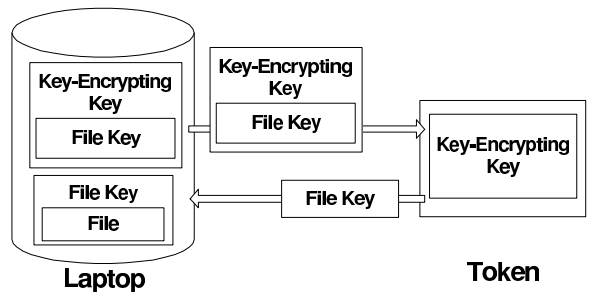


This figure shows the process for authenticating and interacting with the token. Once an unlocked token is bound to a device, it negotiates session keys and can detect the departure of the token.

Figure 1. Token Authentication System

To ensure explicit consent, our model provides for the *binding* of tokens to devices. Binding is a many-to-many relationship; I might interact with any number of devices, and any number of users might share a device. Binding requires the user’s assent but can be long-lived. This limits the usability burden. The binding process requires mutual authentication between device and token; this introduces the question of device and token naming, which we have not yet addressed.

Unfortunately it is possible for a user to lose a token. Token loss is a serious threat, as tokens hold authenticating material; anyone holding a token can act as that user. To guard against this, users must periodically authenticate to the token. This authentication can be persistent, on the order of days. Nominally, any authenticating material in the token is encrypted by a user-supplied password; when the authentication period expires, the token flushes any decrypted material, and will no longer be able to authenticate on the user’s behalf. Placing authentication material in PIN-protected, tamper-resistant hardware [15] further strengthens the token in the event of loss or theft. The transient authentication process, illustrating all of these principles, is shown in Figure 1.



This figure illustrates the process of file key acquisition. Encrypted file keys are read from disk and shipped to the token. The token decrypts it and returns the file key. All laptop/token communication is protected by a session key, negotiated at bind time.

Figure 2. Decrypting File Encrypting Keys

3 Example: Protecting File Systems

To validate this approach, we have built ZIA, a cryptographic file system employing transient authentication. This prototype demonstrates that transient authentication can protect resources against theft or loss without compromising performance or hampering usability. It is explained in detail elsewhere [3]; we summarize it here. ZIA is implemented as a stackable file system layer [6], utilizing the FiST framework [17].

Tie Capabilities to Users Each on-disk object in ZIA is encrypted with some particular file key, K_f . File keys are assigned per-directory. Since the token cannot provide bulk decryption services, the file key is also stored on disk encrypted by some key-encrypting key, K_k . A file that is shared has its corresponding K_f encrypted by more than one K_k . Tokens hold each applicable K_k ; they are never revealed. Key exchanges are illustrated in Figure 2.

Do No Harm When ZIA receives a read request for a file block, it must decrypt it with the corresponding K_f . If it is not already cached, the encrypted version, $K_k(K_f)$, is sent to the token, which decrypts and returns it. Key acquisition is overlapped with reads, and decrypted keys are cached for later reuse. With these optimizations, ZIA adds an overhead of under 10% for a modified Andrew Benchmark, and just over a factor of two for bulk data transfer. In both cases, the overheads imposed by ZIA are indistinguishable from those of Cryptfs [16], a cryptographic file system with a single key in effect for all files. In other words, the overheads are limited by cryptography, not key acquisition.

Secure and Recover on People Time On token departure, ZIA encrypts each cached file block, and flushes each cached K_f . ZIA does not evict encrypted, cached file blocks, but the OS may later evict them of its own accord. On the largest buffer cache we can observe on our

```

typedef int32_t ta_appid_t;

enum lifetime {TA_LIFE_SESSION, TA_LIFE_PERM};

typedef struct ta_keyname_t {
    char* username;
    char* appname;
    char* keyname;
    enum lifetime life;
} ta_keyname_t;

int ta_initialize();

void ta_shutdown ();

/* Registers an application with the library */
ta_appid_t ta_application_reg(IN char* appname,
                             IN char* username);

```

Figure 3. Library Management Functions

hardware—an IBM ThinkPad 570 with 128 MB of memory, running Linux 2.4.10—this process takes less than five seconds. The window of vulnerability is short enough to foil physical possession attacks. As soon as the token enters radio range, ZIA re-fetches each K_f and decrypts all cached, encrypted file blocks. On our hardware, this process takes less than six seconds. ZIA imposes minimal overhead and preserves usability, giving the user no reason to disable it.

Ensure Explicit Consent Tokens only decrypt file keys for laptops to which they have been bound. When a newly-encountered laptop first asks the token for a key, the token passes this request up to its wearer. The user then decides whether this is a request from a legitimate device. If it is not, the laptop will be ignored thereafter. If it is legitimate, the token and laptop use the Station-to-Station protocol [4] to provide both mutual authentication and session key exchange. This session key is used to protect file key traffic between the two devices until the user departs. When the user departs, the session key is dropped. On return, the prior binding remains in force, so Station-to-Station can negotiate a new key without user involvement. Bindings have a long but finite lifetime, on the order of days.

4 Protecting Services and Applications

It is straightforward to provide transient authentication services to a file system; the semantics of identity and privilege are well-defined, and the implementation is controlled by the operating system. Most applications inherit these same notions of user identity from the operating system. Transient authentication can be extended to applications by protecting the virtual memory space of each process.

By default, the system protects every process on the machine, except essential kernel threads and processes related to token communications. During the boot process, the token and machine agree upon an encryption key. Upon user departure, the operating system suspends each processes and masks arriving signals. The OS then encrypts each in-memory page belonging to the process and erases the key.

When the user returns, the system fetches the decryption key from the token and restores each page. The processes are restarted and the system continues without loss in performance. Combined with swap space protection [13], this mechanism fully protects virtual memory. We are currently implementing address-space protection.

There are two reasons why this brute-force approach may not be desirable. First, completely suspending all applications—even those without sensitive data—is effective but indiscriminate. Second, not all applications inherit their notions of identity and authentication from the operating system. For example, a user browsing the web may interact with dozens of different services, each with its own user name.

We are developing an API for applications to make use of transient authentication services directly. This API includes management functions, shown in Figure 3 and token interaction functions, shown in Figure 4. The initialization and shutdown functions are used to initiate connections to a per-machine service. The registration functions inform this service of the application’s use of the token. This service can then control application requests, including logging and caching of results.

Applications that cache keys—or the decrypted information that they protect—must be informed whenever the user

```

/* Registers/Unregisters callback functions */

int ta_auth_loss_reg(IN ta_auth_hdlr_t hd,
                    IN void *data);

int ta_auth_loss_unreg(IN ta_auth_hdlr_t hd,
                      IN void *data);

int ta_auth_gain_reg(IN ta_auth_hdlr_t hd,
                    IN void *data);

int ta_auth_loss_unreg(IN ta_auth_hdlr_t hd,
                      IN void *data);

/* Decrypt a key with the token */
int ta_decr_buf (IN ta_keyname_t *key_name,
                IN const void *in_key,
                IN size_t len, OUT void ** out_key);

```

Figure 4. Token Interaction Functions

departs or returns. To do so, the application registers a callback: the name of the function to call in the event of departure or return. The transient authentication system polls the user token, and calls each registered callback in the event of a change. To access sensitive resources, the application must first acquire a decryption key. Applications store an encrypted version of the key and then use the token to decrypt it. When the application needs to create a new key, a random encrypted key can be created and “decrypted” using the token.

Unfortunately, many applications assume that authentication need only be checked once or, at best, infrequently. For example, SSH authenticates users only when initiating a remote login; the connection remains in force until explicitly closed. There must also be a mechanism to ensure that applications respond to departure notifications in a timely way; those that do not will be subject to the more drastic step of full address-space protection.

One example application that we have modified is the open-source, Mozilla web browser; we are currently working on extending support to other applications. Mozilla, and web browsers in general, store sensitive information in numerous places. So far we have identified the browser cache, password manager, SSL key store, certificate store, and the cookie store as the most critical. The use of programming language tools to find such sensitive material is an important open problem.

After registration with the API, Mozilla uses the token to generate fresh keys for each of these resources. Although the actual implementation for each of the secrets is slightly different, each remains encrypted when the user is

not present. While the user is present, the browser has the capability to decrypt this information and use it. Otherwise, requests return an error message to the user interface.

5 Related Work

Several efforts have used proximity-based hardware tokens to detect the presence of an authorized user. Landwehr [7] proposes disabling hardware access to the keyboard and mouse when the trusted user is away. This system does not fully defend against physical possession attacks. At the very least, the contents of disk and possibly memory may be inspected at the attackers leisure. Similar systems have reached the commercial world. For example, the XyLoc system [14] could serve as the hardware platform for our authentication token. Our contribution is not the use of a hardware token for proximity detection. Rather, it is the principle of transient authentication and the way in which it affects system and application design.

We have rejected the use of smartcards for authentication services [2]. There are two ways to use smartcards, insertion and swiping. Inserted cards are likely to be left in the machine when the user is away. Swiping must be done frequently, or employ long timeout periods. In either method, smartcards have the same weaknesses as passwords.

One could imagine using biometric authentication rather than a worn, wireless token to provide proximity cues. Unfortunately, biometrics suffer from several usability problems. They have a large false negative rate [12], and are not easily revocable—if someone has a copy of your thumbprint, you cannot easily change it. Furthermore, bio-

metric authentication often requires some conscious action on the part of the user. The one exception is iris recognition [10], but this scheme requires three separate cameras, a bulky and expensive proposition for mobile devices.

There are a number of file systems that provide transparent encryption: Blaze's CFS [1], Zadok's Cryptfs [16], and Microsoft's EFS [8]. None of these tie user authentication to the encryption process properly. Some systems, such as EFS, require the user to reauthenticate after certain events to bound the window of vulnerability. This increases security, but decreases usability.

6 Conclusion

Computing systems currently depend on persistent authentication, in which user authentication is assumed to hold for a long, perhaps unbounded, time. Given the poor physical security afforded by mobile or ubiquitous devices, this is untenable. Instead, we propose the use of transient authentication, in which a user wears a small token that constantly authenticates on his behalf. Transient authentication secures systems against physical possession attacks without compromising performance or usability. We have constructed a cryptographic file system using this approach, and are currently constructing an API to provide support for a broad number of applications and services.

Acknowledgements

The authors wish to thank Peter Chen, who suggested the recovery time metric, and Peter Honeyman, for many valuable conversations about this work. Erez Zadok's FiST framework substantially simplified the construction of ZIA. This work is supported in part by the Intel Corporation; Novell, Inc.; and the Defense Advanced Projects Agency (DARPA) and Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Intel Corporation; Novell, Inc.; the Defense Advanced Research Projects Agency (DARPA); the Air Force Research Laboratory; or the U.S. Government.

References

- [1] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conf. on Computer and Communications Security*, pages 9–16, Fairfax, VA, November 1993.
- [2] M. Blaze. Key management in an encrypting file system. In *Proceedings of the Summer 1994 USENIX Conference*, pages 27–35, Boston, MA, June 1994.
- [3] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proceedings of the ACM International Conference on Mobile Computing and Communications*, Atlanta, GA, September 2002. to appear.
- [4] W. Diffie, P. van Oorschot, and M. Wiener. *Design Codes and Cryptography*. Kluwer Academic Publishers, 1992.
- [5] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. Internet Draft, March 1996.
- [6] J. S. Heidmann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [7] C. E. Landwehr. Protecting unattended computers without software. In *Proceedings of the 13th Annual Computer Security Applications Conference*, pages 274–283, San Diego, CA, December 1997.
- [8] Microsoft. Encrypting File System for Windows 2000. <http://www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp>.
- [9] C. Narayanaswami and M. T. Raghunath. Application design for a smart watch with a high resolution display. In *Proceedings of the Fourth International Symposium on Wearable Computers*, pages 7–14, Atlanta, GA, October 2000.
- [10] M. Negin, T. A. Chemielewski Jr., M. Salganicoff, T. A. Camus, U. M. Cahn von Seelen, P. L. Venetianer, and G. G. Zhang. An iris biometric system for public and personal use. *IEEE Computer*, 33(2):70–5, February 2000.
- [11] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.
- [12] P. J. Phillips, A. Martin, C. L. Wilson, and M. Przybocki. An introduction to evaluating biometric systems. *IEEE Computer*, 33(2):56–63, February 2000.
- [13] N. Provos. Encrypting virtual memory. In *Proceedings of the Ninth USENIX Security Symposium*, pages 35–44, Denver, CO, August 2000.
- [14] Ensure Technologies. <http://www.ensuretech.com/>.
- [15] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workshop of Electronic Commerce*, pages 155–70, New York, NY, July 1995.
- [16] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.
- [17] E. Zadok and J. Nieh. FiST: a language for stackable file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 55–70, San Diego, CA, June 2000.