Protecting Applications with Transient Authentication

Abstract

How does a machine know who is using it? Current systems authenticate their users infrequently, and assume the user's identity does not change. Such *persistent authentication* is inappropriate for mobile and ubiquitous systems, where associations between people and devices are fluid and unpredictable. We solve this problem with *Transient Authentication*, in which a small hardware token continuously authenticates the user's presence over a short-range, wireless link. We present the four principles underlying Transient Authentication, and describe two techniques for securing applications. Applications can be protected transparently by encrypting in-memory state when the user departs and decrypting this state when the user returns. This technique is effective, requiring just under 10 seconds to protect and restore an entire machine, but indiscriminate. Instead, applications can utilize an API for Transient Authentication, protecting only sensitive state. We describe our ports of three applications—PGP, SSH, and Mozilla—to this API. Mozilla, the most complicated application we have ported, suffers less than 4% overhead in page loads in the worst case, and in typical use can be protected in less than 250 milliseconds.

1 Introduction

How does a device know that the right person is using it? 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.

Authentication requires that a user supply some proof of identity—via password, smartcard, or biometric—to a device. Unfortunately, it is infeasible to ask users to provide authentication for each request made of a device. Imagine a system that requires the user to manually compute a message authentication code [21] for each command. The authenticity of each request can be checked, but the system becomes unusable. Instead, users authenticate infrequently to devices. User authentication is assumed to hold until it is explicitly revoked, though some systems further limit its duration to hours or days. Regardless, in this model authentication is persistent.

Persistent authentication creates tension between security and usability. To maximize security, 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, a user wears a small token, such as the IBM Linux watch [18], 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.

Transient Authentication has been applied to cryptographic file systems [6] and could be extended to protect swap space [23]. These provide a good first line of defense, protecting persistent storage from physical possession attacks. Unfortunately, they do not protect applications. An application that reads data from a cryptographic file system—or receives data from a secure network connection [1, 29]—holds that data in memory without protection. This paper extends Transient Authentication to address this vulnerability.

We first describe the trust and threat model we consider, and enumerate the four principles underlying Transient Authentication. We then present two mechanisms for protecting in-memory application state.

The first, *application-transparent protection*, provides protection within the kernel. When the user departs, all user processes are suspended and in-memory pages encrypted. When the user returns, pages are decrypted and processes restarted. Protection and recovery processes each take at most ten seconds on our hardware, and applications need not be modified to benefit from this service.

Application-transparent protection is effective but indiscriminate. There are processes that can safely continue in the user's absence, and a few processes may be

able to selectively identify and protect sensitive state. Our second mechanism, *application-aware protection*, supports such applications. We provide an API for applications to use Transient Authentication services directly. We have modified three applications—PGP, SSH, and Mozilla—to make use of this API. In exchange for such modifications, these applications can be protected and restored in well under half a second, and suffer no noticeable degradation of run-time performance.

2 Trust and Threat Model

Our focus is to defend against attacks involving *physical possession* of a device or *proximity* to it. Possession enables a wide range of exploits. The easiest attack is to use authentication credentials that are cached by the operating system or individual applications. Even without cached credentials, console access admits a variety of well-known attacks; some of these result in root access. A determined attacker may even inspect the memory of a running machine using operating system interfaces or hardware probing.

Transient Authentication must also defend against observation, modification, or insertion of messages sent between mobile devices and the token. Simple attacks include eavesdropping in the hopes of obtaining sensitive information. A more sophisticated attacker might record a session between the token and laptop, and later steal a misplaced laptop in the hopes of decrypting prior traffic. We defeat these attacks through the use of well-known, secure protocols [10, 21].

Transient Authentication's security depends on the limited range of the token's radio. Repeaters or arbitrarily powerful transmitters and receivers could be used to extend this range. This is sometimes called a wormhole attack [13]. The rapid attenuation of high frequency radio signals makes attacks using powerful transmitters difficult in practice. A better solution would use timing information to detect the distance of the token from the device. This technique has been proposed by Brands and Chaum [3] and explored in the Wormhole detection project [13], though neither has built a practical implementation.

Transient Authentication does not defend against a trusted but malicious user who leaks sensitive data. It also does not consider network-based exploits to gain access to a machine, such as buffer overflow attacks [7]. Finally, we do not protect against denial of service attacks that jam the spectrum used by the laptop-token channel.

3 Transient Authentication Principles

Transient Authentication is governed by a set of four guiding principles. First, users must hold the sole means to access sensitive resources or invoke trusted operations on the device. Second, the system must impose no additional usability or performance burdens. Third, the mechanisms to secure and restore sensitive data do not need to be faster than people using them. Fourth, users must give explicit consent to actions performed on their behalf. This section expands on each of these principles.

3.1 Tie Capabilities to Users

The ability to perform sensitive operations must ultimately reside with the user rather than her devices. For example, the keys that decrypt private data must reside on the user's token, and not on some other device.

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 her token) leave, and the master capability cannot be exposed beyond the token.

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 memory access control. The operating system can be forced to provide the contents of physical memory through direct OS interfaces such as Linux's `/dev/mem` and Windows' `\Device\PhysicalMemory`. An encrypted memory store, with the keys stored only on the token, is not subject to the same attack.

3.2 Do No Harm

Investing capabilities with users increases the security of the system. However, increases in security cannot impose additional burdens. When faced with inconvenience, however small, users are quick to disable or work around security mechanisms. Both performance and usability must remain unaffected.

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 can impose no usability constraints beyond those of 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) [11] protocol requires processing time for encryption and authentication. However, this cost is masked by network latency.

3.3 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 quickly. However, since people are slow, the limit is on the order of seconds, not milliseconds.

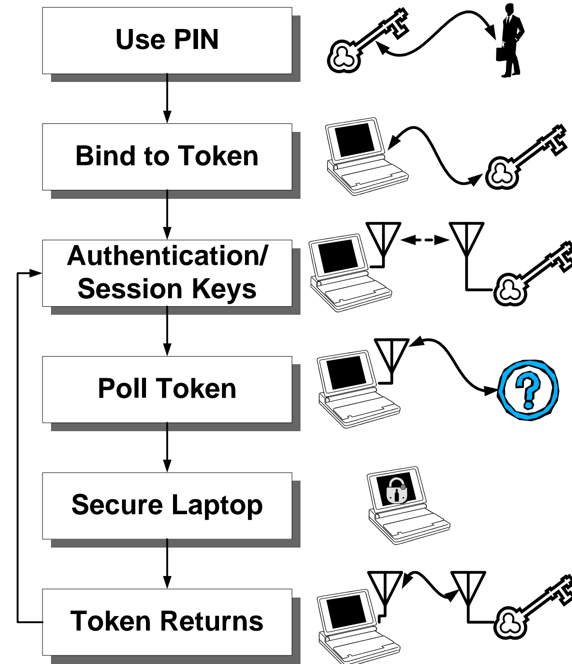
Rather than simply erasing sensitive information, one might prefer to encrypt and retain it. This additional work can save time on restoration: when the user returns, one can obtain the proper key from the token and decrypt the data in place, restoring the machine to pre-departure state. Since the restoration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

3.4 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 another user's token to provide capabilities, 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.

Theoretically, users could confirm every capability requested by the device. However, usability is paramount, so the granularity of authorization must be much larger. Instead of an action-by-action basis, user consent is given periodically on a device-by-device basis.

To ensure explicit consent, our model provides for the *binding* of tokens to devices. Binding is a many-to-many relationship; One might interact with any number of devices, and any number of users might share a device.



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

Binding requires the user's assent but can be long-lived, limiting the usability burden. The binding process requires mutual authentication between device and token.

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 many 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 [28] 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.

4 Application-Transparent Protection

Applications store sensitive information, such as credit card numbers and passwords, in their virtual address space. Even with an encrypted file system [6] and swap space [23], the in-memory portions of an applications address space vulnerable to attack. The memory

bus or chips may be probed by a knowledgeable attacker, or OS interfaces can be exploited to examine raw memory contents.

This section describes a technique, called application-transparent protection, for protecting in-memory process state. The main benefit of this technique is that it protects processes *without* modification. The application designer does not need to identify which data structures contain secret data and users do not have to designate which processes to protect.

4.1 Design

Applying the first stated goal of Transient Authentication requires that the capability of reading memory be tied to the user. One approach would be to require each load and store to use encryption, using keys only available on the token. The performance of the machine would suffer greatly, clearly violating the principle of "do no harm". An alternative would be to protect the machine by flushing the contents of memory into the swap space and zeroing the memory whenever the user departs. This scheme would make use of swap space encryption, combined with keys available only on the token. On return, the paged-out memory would be read back from the disk into the memory pages. Unfortunately, both protecting and maintaining the machine would require a significant amount of overhead in disk operations, leaving the machine vulnerable longer and burdening the user. This would violate the principle of "securing and restoring on people time".

Instead, the system must encrypt the virtual memory of processes in place. Since all the encryption operations are done in memory, this mechanism provides both fast protection and recovery. To avoid corrupting the encrypted memory, processes must first be placed in a hibernation state, preventing them from executing while the user is away. Certain processes can be designated as unprotected, but most processes will not execute until the user returns. On recovery, the memory is decrypted and the process is re-animated; to a returning user it appears as if nothing has changed.

We have found that the recovery process is fast enough to remain unnoticed by users. However, if the ratio of memory size to processing speed were much greater than on our test machine, it is possible that the securing or recovery process could be too lengthy. If the securing process is too slow, the application-aware techniques presented later in the paper will be required. If recovery is the bottleneck, it is possible to first recover applications the user will interact with quickly. Operating systems already track interactive jobs to provide good response time in process scheduling [26], enabling informed selection of recovery order. However, we expect that the current memory/processor balance will continue for the foreseeable future making this technique unnecessary.

4.2 Implementation

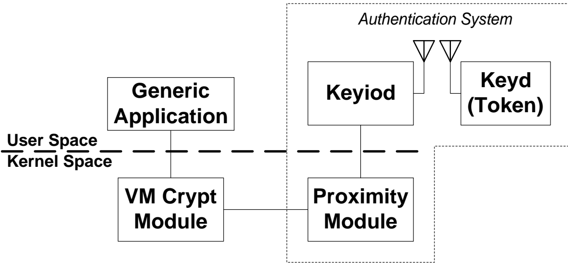
We have built a Linux prototype to protect the in-memory portions of application state. At startup, an in-kernel module receives a fresh key from the token to govern the memory of running processes. The module receives notifications of the token's status from another in-kernel module. When it receives notification of user departure, each processes is set to hibernate, using techniques borrowed from the Linux Software Suspend project [5]. First, each process is marked as hibernating and also as having a pending signal. The only processes allowed to continue running are essential tasks related to Transient Authentication and the operating system. The processes are woken up and the kernel signal dispatcher prevents the process from running until the hibernate flag is cleared. This has the property of having to wait for uninterruptible processes to become interruptible. However processes normally last in this state only for a short time. It may be the case that a buggy process has become stuck in an uninterruptible state; we are currently unable to handle this situation.

After hibernation is complete, the module walks the virtual memory space of each process, looking for in-memory pages. Each in-memory page is encrypted using the pre-fetched key, and marked as such to prevent multiple encryptions of shared memory pages. The decrypted copy of the key is then thrown away. On user return, the process is reversed—the kernel fetches a decrypted version of key from the token, the memory is decrypted and all processes are awoken from where they left off.

Free memory pages present a special difficulty. Applications may have allocated memory, stored secret information in that space, and then terminated. This memory is returned to the OS, and it may still contain remnants of that information. To protect these remnants, the module must walk the list of free pages and zero the memory of each page in the list.

An overview of the transparent protection system is shown in Figure 2. Fetching the encryption key from the token is handled by a pair of user space daemons, `keyiod` on the laptop and `keyd` on the token, communicating via a wireless link. Exposure of the virtual memory encryption key would nullify its protections, so each message between `keyiod` and `keyd` must be encrypted. Further, since the token is used to create fresh encryption keys, the link must be authenticated as well.

Mutual authentication can be provided with public-key cryptography [19]. In public-key systems, each



This figure shows the components in the transparent protection system. When authentication is lost, a kernel module encrypts the in-memory state of any generic application. Authentication and token communication are handled by a kernel proximity module and a user space daemon.

Figure 2. Transparent Protection

principal has a pair of keys, one public and one secret. To be secure, each principal's public key must be certified, so that it is known to belong to that principal. Because laptops and tokens fall under the same administrative domain, that domain is also responsible for certifying public keys. `Keyiod` and `keyd` use the Station-to-Station protocol [10], which combines public-key authentication and Diffie-Hellman key exchange.

Each message includes a *nonce*, a number that uniquely identifies a packet within each session to prevent replay attacks [4]. In addition, the session key is used to compute a *message authentication code*, verifying that a received packet was neither sent nor modified by some malicious third party [21].

The kernel cryptographic module must be informed when the token is no longer present. To provide this notification, we add a periodic challenge/response between the laptop and the token. These proximity polling messages are generated by a second module in the kernel. We currently set the interval to be one second; this is long enough to produce no measurable load, but adds little to the amount of time needed to protect the laptop.

5 Application-Aware Protection

Transparent application support is an effective technique, but an indiscriminate one. There are several disadvantages in protecting every process on the machine, regardless of the sensitivity of their contents. A process that only occasionally conducts sensitive operations must be completely stopped, regardless of its current tasks. Certain processes could be statically designated as non-sensitive, or the process could mark itself as sensitive dynamically. However, if two processes communicate through shared memory, both must be stopped, even though only one may be sensitive. Also, some applications that depend on constant input or network traffic may not survive the hibernation process. This burdens the user, who must either restart those applications or perform work to restore the previous state.

To combat these shortcomings we provide an interface for an application to manage its own sensitive information. This allows greater flexibility in handling loss of authentication and permits the application to continue to run regardless of authentication state. In order to provide this capability, we have designed an application programming interface, or API, hat allows applications to use Transient Authentication services. Applications must be restructured to depend on capabilities, such as keys, held by the token. For performance, these capabilities can be cached, but they must be flushed when the token leaves.

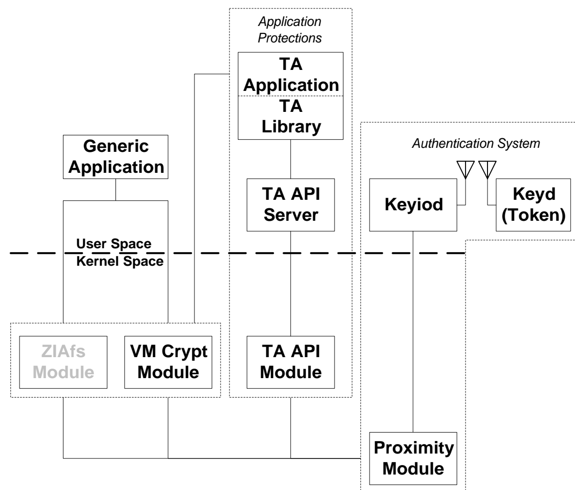
Some applications and services already manage authentication and access to sensitive resources. Most of these systems revoke access through either explicit user logout or expiration of a long-lived session. Some of these applications and servers may also provide various levels of service, depending on the specific credentials of the user. Such applications already manage identity and privilege, and would benefit from direct use of Transient Authentication services.

An overview of the system is shown in Figure 3. Generic applications can take advantage of Transient Authentication using transparent protection. Modified applications are compiled with a Transient Authentication library and communicate with the kernel using a user-space server. All interactions with the token pass through the proximity polling module and a user-space communication daemon. We have implemented parts of the system in the kernel to make the system fail-safe. If any part of the system fails, the application should still receive a notice of authentication loss.

5.1 Protecting Targeted Secrets

Identifying secret data is the most difficult part of protecting an application. The application designer must first consider the threat model and user requirements. For instance: Is all of the user's data secret? What about the meta-data? What about data received from the network? For example, the text of a word processor document is probably private, the formating of that document may or may not be, and the state of local program variables is probably not. There are no hard rules for determining these classifications and it must be left to the designer of the application.

Once secrets have been identified, we use two different mechanisms to tie capabilities to the token. One is to encrypt secrets when the user leaves and forget the locally stored copy of the key. When the user returns, that key can be retrieved from the token and the secret



This figure shows the various components used in the Transient Authentication system. Generic applications can be protected by the virtual memory encryption system and the ZIA file system. Modified applications are compiled with a Transient Authentication library and communicate with the kernel through a user-space server. All communications with the token go through a proximity polling module and a user-space communication daemon.

Figure 3. TA Components

decrypted. Another choice is to always store the information encrypted, and decrypt it for every short term use.

Choosing which mechanism to use depends on the properties of the data, including size and frequency of use. Accessing secrets must not take a noticeable amount of time, and protection and restoration must be done in "human time". In some cases, both of the mechanisms will each conform to the principles of Transient Authentication, allowing the programmer to pick the more convenient option.

5.2 Application Programming Interface

Before a user starts an application that employs the Transient Authentication API, that user must have one or more *master keys* for that application installed on their token. In our implementation, master keys are 128-bit AES [8] keys. These keys must be installed by an administrative authority, and can never be exposed beyond the token. As we will see, the master key is typically used as a *key-encrypting key*, but can sometimes protect small data items directly. Once a key is installed, the API is available. It is summarized in Figure 4.

On startup, each protected application *registers* itself with the API, providing the its name and the user running it. It then installs a *handler*. The handler is called when the token fails to respond to a request, revoking authentication, or when a departed token once again is in range, reestablishing authentication.

Each master key acts as the capability to perform sensitive actions on behalf of its user and application. Simple examples of such actions are reading cached passwords or credit card numbers. These items are small; it is feasible to ship encrypted copies of them to the token, decrypt them, and send them back. This can be done directly with `ta_encr_buf` and `ta_decr_buf`. The application may decrypt and cache such items, but must clear them when notified of token departure.

```
/* Register an application with the library */
int ta_application_reg (IN char* app_name,
                        IN char* username);

typedef
enum ta_change{TA_LOSS, TA_GAIN} ta_change_t;

typedef
int (* ta_auth_hdlr_t ) ( IN ta_change_t change,
                          IN int flags );

/* Register a handler for change in
   authentication */
int ta_auth_change_reg (IN int appid,
                        IN ta_auth_hdlr_t hdlr );

typedef char* ta_keyname_t;

/* Decrypt a buffer on the token with a key */
int ta_decr_buf (IN int appid,
                 IN ta_keyname_t keyid,
                 IN char* inbuf,
                 IN size_t inlen,
                 OUT char** outbuf,
                 OUT size_t* outlen );

/* Encrypt a buffer on the token with a key */
int ta_encr_buf (IN int appid,
                 IN ta_keyname_t keyid,
                 IN char* inbuf,
                 IN size_t inlen,
                 OUT char** outbuf,
                 OUT size_t* outlen );
```

This listing shows the API for Transient Authentication. Three types of functions are included: registration with the user-space server, registration of authentication callback functions, and buffer decryption using the token and previously registered key.

Figure 4. Transient Authentication API

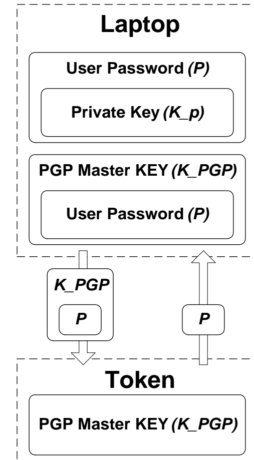
Some things cannot be handled with direct encryption and decryption. Passing large data elements directly to the token for decryption would likely impose too great of a performance penalty. To protect large elements, the application must first create a *sub-master key*. Sub-master keys cover large objects. Encrypted copies of the sub-master can be stored at any time, while decrypted copies can be kept only while authentication holds. Our idiom for creating sub-master keys is to choose a random number as the encrypted key, and have the token "decrypt" it. This ensures that a secret key is never generated without the token's involvement.

On startup, applications do not hold any sensitive data; they must first either decrypt an item or obtain a derived key. These decryption requests will fail if the token is out of range, leaving the application in a safe state. Once the first item or key is successfully decrypted, the user is considered authenticated. Thereafter, the run time system tracks the token's comings and goings, reporting them to registered handler. The next three sections describe how three user applications were modified to use this API.

5.3 Pretty Good Privacy (PGP)

Pretty Good Privacy [1], or PGP, uses the RSA asymmetric encryption algorithm to digitally sign and encrypt data. Users possess a pair of keys, one public and one private. Data can be encrypted using the public key and only someone who knows the private key can decrypt it. The private key can also be used to sign the message, and anyone can verify the signature using the public key. PGP can be used to provide data integrity and privacy to a great variety of applications, however we will focus on email.

The most valuable secret held by PGP is the user's private key K_p. Commonly, K_p is protected by a user's password, P, denoted as $P\{K_p\}$. When using an email client, such as Pine, the user is prompted for the password on each signature or decryption operation. In adding Transient Authentication services to PGP we have chosen to preserve the original semantics of the application and minimize modifications. To do this we have protected K_p with a random password, P, encrypted by a key on the token, K_{PGP}. This chain of keys is written as $K_{PGP}\{P\}, P\{K_p\}$. The modifications made to PGP are summarized in Figure 5. When a user asks PGP to decrypt or sign a piece of email, the private key, K_p, is required. PGP reads both $K_{PGP}\{P\}$ and $P\{K_p\}$ from the user's PGP key directory. It sends a decryption request to the token containing $K_{PGP}\{P\}$ and the token returns P. P is used to decrypt K_p and is then thrown away. The signing or decryption process uses K_p for as long as the operation takes, and the token is no longer needed.



This figure illustrates the modifications made to PGP. The private key, K_p of the user is protected by a password, P. P is encrypted by K_{PGP}, which is only known to the token. Each time PGP needs to use K_p it asks the token to decrypt P, which enables the laptop to decrypt K_p.

Figure 5. PGP Modifications

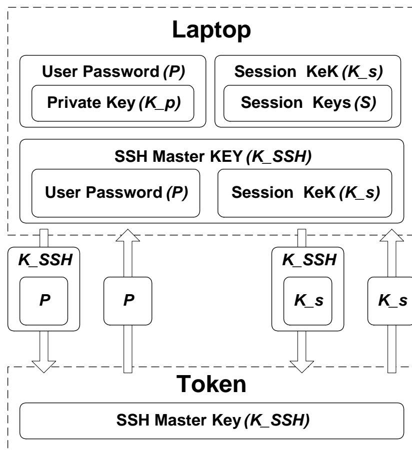
Email encryption and decryption is a short process. To keep the modifications to PGP as simple as possible, any loss of authentication while using the private key causes the process to exit. Any secrets contained in freed memory can be protected by the zeroing of free pages in the transparent protection kernel.

A mail program, such as `pine`, must employ PGP's output with care. For instance, if decrypted messages are displayed to the screen, the mailer must take steps to obscure that data upon loss of authentication. One possible mechanism would be to reset the display to the message index. Another option would be to redisplay the encrypted form of the message and recover the decrypted version when the user returns.

5.4 OpenSSH

The Secure Shell [29] suite of tools provides authenticated and encrypted equivalents for `rsh` and `rcp`, called `ssh` and `scp`. Client applications authenticate servers based on public key cryptography. Servers authenticate users based on passwords or public keys. Data transmitted during the session is encrypted using a key exchanged in the authentication stage. We have modified an open-source secure shell, OpenSSH; a summary of the modifications is shown in Figure 6.

OpenSSH contains two secrets that need protection, the private key, K_p, used for authentication, and the session key, S, used to encrypt data. The private key is covered by the same methods as PGP—the password, P, for K_p can be decrypted by the token's K_{SSH}.



This figure illustrates the modifications made to OpenSSH. The user's private authentication key is protected by a password P, which is encrypted by a key K_{SSH}. When the user is not present, the session keys, S, are encrypted by a session key encrypting key K_s, which is encrypted by K_{SSH}, as well. When OpenSSH needs to authenticate, it uses the token to decrypt P, giving it access to K_p. Similarly, when the user returns, the token is used to decrypt K_s, giving access to the session keys.

Figure 6. OpenSSH Modifications

The authentication phase generates the session key, S, which is cached. Before the session continues, OpenSSH must protect the session key. First, OpenSSH creates a new "encrypted" key, $K_{SSH}\{K_s\}$. It then uses the token to decrypt the encrypted key, yielding K_s. Finally, OpenSSH uses K_s to create an encrypted version of the session key, denoted $K_s\{S\}$, which it caches.

While the user remains present, S remains decrypted in memory for session encryption and decryption. If a disconnection notification is received, OpenSSH flushes both S and K_s, but retains $K_s\{S\}$ and $K_{SSH}\{K_s\}$. When the user returns, OpenSSH must decrypt K_s using the token. It can then decrypt S and continue the session.

Each use of the session key requires a simple check that S is still available. This check takes a small amount of time, slowing data transmission by a negligible amount. If S is encrypted, the transmission of data blocks, and received data is held in the network buffer—still encrypted—until the user returns. Any blocked sessions are resumed where they left off. It may be possible for unencrypted data to get passed between the terminal and SSH after a disconnection. We are currently working on methods to prevent this from happening, such as locking the keyboard first, rejecting all data from the terminal, or returning an error to the pipe.

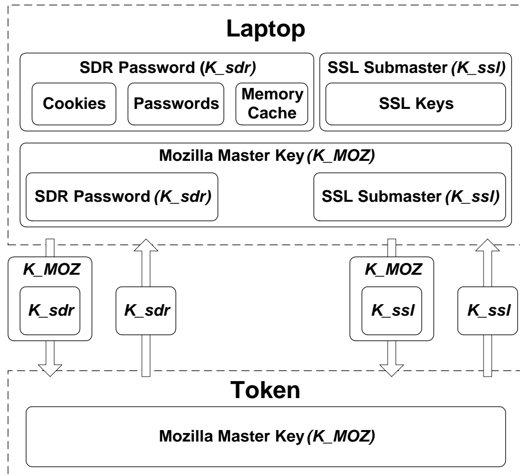
5.5 Mozilla Web Browser

Web browsers provide secure access to online accounts, e-commerce, and web-based email. Consider a typical session for accessing a secure web server at a bank. First, the browser creates a Secure Socket Layer (SSL) session with the bank's server. SSL provides session encryption to an authenticated server. The user authenticates himself by typing an account number and password into a web form. The browser often caches this information to make future logins easier. The server then sets a cookie on the user's local machine to authenticate future requests during this session. Note that SSL can provide for client authentication, but the vast majority of sites use cookies instead. Web pages, such as an account statement, can now be retrieved from the server and remain available in the browser's cache. This example identifies several places where secret information resides in the browser's address space: SSL session keys, cached passwords, cookies and the cache of the browser.

We have added Transient Authentication to the Mozilla web browser. Mozilla is a large and complex piece of software, containing more than 250MB of code and using several different programming languages. Some effort was made in the original source code to separate confidential and non-confidential data; however, this mostly pertained to secret keys themselves and not to sensitive data such as cookies and the cache. Mozilla also includes a module, the Secret Decoder Ring (SDR), that can be used to encrypt and decrypt arbitrary data. The SDR module depends on a user login to explicitly provide a decryption key. This provides an ideal location to add Transient Authentication to the system. SSL keys are contained in the same module as SDR, and therefore it uses these internal encryption functions, rather than the external interface. A diagram of the components in the modified browser is shown in Figure 7.

SSL session keys are used frequently, so it would be inefficient to decrypt them on every use. Instead they remain decrypted until a token departure; they are then encrypted in-place. SSL session keys could be flushed and recreated when the user returns, however to replicate the current semantics we keep the SSL session open.

Cached passwords are used very infrequently and can be stored on disk. In this case, it makes sense to have SDR decrypt the information each time it is used—Mozilla already has this capability. Cookies are used more frequently than stored passwords, but less than SSL keys. Thus, either method could be used. We have chosen to leave them encrypted and decrypt them using SDR on each use. The evaluation presented in Section 6.4 shows that this overhead is tolerable.



This figure depicts the modifications made to the Mozilla web browser. Cookies, passwords and the memory cache, all depend on Mozilla's Secret Decoder Ring for encryption and decryption. The password for the ring can be accessed using the token. SSL operates in the same way, and the sub-master key used to encrypt SSL keys can be obtained using the token.

Figure 7. Modified Mozilla Components

The web cache is split into two parts, an in-memory cache and an on-disk cache. Mozilla's policy is to store data from SSL connections only in memory and never on disk. All non-SSL data is considered to be previously exposed on the network and is not protected, although there is nothing that precludes protecting this via file system encryption. The items in the cache are potentially large in size and frequently accessed. However, the cache is of limited size and can be encrypted in bulk very quickly. Thus, each item in the cache retrieved from SSL connections is SDR-encrypted on user departure and decrypted on user return. The password for SDR is erased when the user leaves and retrieved from the token when the user returns.

5.6 Application-Aware Limitations

After making the modifications to these applications we have noted several limitations. First, sensitive data may no longer be reachable in the application. These include secrets contained in leaked memory and memory that has been freed. It is not possible to protect the former. However, using a pre-loaded library, calls to `realloc`, `free`, and `delete` can be intercepted and modified to zero freed memory.

Second, if the application has written secret information to the screen in a readable form, the application itself must directly obscure it; it can overwrite it with blank pixels or other non-protected information. More generally, any secrets that have been passed to other processes may not be protected if they do not employ the API as well. We are currently looking at adding application-aware support to windowing systems, window managers, and interface toolkits.

The third difficulty is the most challenging: identifying the secrets in the application. In the examples, we have made an effort at identifying data structures containing secret data. However, this is an ongoing process that improves as we learn more about the structure of these programs. Since the modifications were not made by the original author of the applications, the effort is possibly more error-prone. In particular, if the application has made a copy of secret data that was not noticed during our examination, it will not be protected. We are currently looking at methods to analyze the flow of secrets in the memory space. One alternative may be to use language support [17].

6 Evaluation

In evaluating Transient Authentication, we set out to answer the following questions:

- What overhead does Transient Authentication impose on the system?
- Can Transient Authentication secure applications quickly enough to prevent attacks when the user departs?
- Can Transient Authentication recover application state before a returning user resumes work?

To answer these questions, we subjected our prototype to a variety of benchmarks. For these experiments, the client machine was an IBM ThinkPad X24, with 256 MB of physical memory and a 1.1 GHz Pentium III CPU. The token was a Compaq iPAQ 3870 with 64MB of RAM. They were connected by a Bluetooth [16] wireless network running in PAN mode. All encryption, except the authentication phase, is done using AES [8] with 128 bit keys. The token is somewhat more powerful and larger than current wearable devices. However, the rapid advancements in embedded, low-power devices makes this a realistic token in the near future.

6.1 Transparent Protection

Transparent protection has no effect on system performance while the user is present. To measure the cost of protection and recovery we allocated 200MB of memory to a user process, occupying all available physical memory. The machine was also running a standard set of user processes, including a window manager and several shells—a total of 38 user processes not including those used for Transient Authentication and Bluetooth. We

	Time, sec	Over Normal
Normal (small)	0.02 (0.00)	-
With TA (small)	0.11 (0.01)	437%
Normal (large)	20.02 (1.24)	-
With TA (large)	20.06 (0.59)	0.21%

Figure 8. PGP Signing and Encrypting

disconnected the token and reconnected it and measured the time it took to secure and recover the machine. Securing the machine required 632 microseconds to freeze all the processes, 8.92 seconds to encrypt 215.9MB of in-memory state, and 6.00 milliseconds to zero 2.25MB of free pages. On recovery, the system required 7.72 seconds to decrypt the same 215.9MB of state, and 21.2 milliseconds to unfreeze the processes. Thus, the system can encrypt state at 24MB/s, zero pages at 375 MB/s, and decrypt state at 28 MB/s. In total the machine can secure and recover our machine in less that 10 seconds each.

6.2 PGP

We subjected PGP to 50 trials of signing and encrypting two files, one 10kB in size and one 10MB. This is to simulate the two common cases of encrypting small pieces of email and large messages containing attachments. The mean and standard deviation for each experiment are reported in Figure 8.

Recall that Transient Authentication-enabled PGP uses the token only for initial authentication. Therefore the only impact on performance is the additional overhead of using the token to decrypt the private key password. Both large and small files only require a small amount of overhead, although the effects are exaggerated for the otherwise fast operations on short files. In either case, the user is unlikely to notice the difference.

6.3 OpenSSH

The modified OpenSSH uses Transient Authentication for initial authentication and for protection of the session key. To measure the impact on a user's session we used a script to provide a typical user input to an `ssh` session. The script logs into another machine and runs a series of user commands: `pine`, opening a mailbox and a single message, `ls` of the home directory, running `emacs`, a `find` on a small directory, and `logout`. Between each user input there is an additional think time of two seconds. The cost of acquiring the key to login accounts for the majority of the overhead in the typical case. To measure this, we ran a second experiment: logging into a remote machine 20 times and computed the average overhead. A third experiment measures the overhead of checking for a decrypted session key on each key by using scp to copy a 10 MB file across the network for 20 trials. The results for each of these experiments are shown in Figure 9.

	Time, sec	Over Normal (%)
Normal (session)	41.01 (0.09)	-
With TA (session)	41.31 (0.15)	0.72%
Normal (login)	0.47 (0.00)	-
With TA (login)	0.72 (0.03)	52.9%
Normal (scp)	18.96 (3.88)	-
With TA (scp)	19.21 (2.74)	1.31%

Figure 9. SSH Experiments

The results show that typical user sessions are almost unaffected by use of the token—any overhead is dwarfed by think-time and the length of the session. The login micro-benchmark confirms that login accounts for most of the overhead. Long sessions also mask the additional login time, shown by the statistically identical times for modified and unmodified `scp`.

We also want to know how long it takes to secure and restore `ssh` session keys. Each `ssh` session has an incoming key and an outgoing key and each are recovered separately. We instrumented ten disconnections and reconnections of the token. The results show a negligible amount of time needed for protection and 130 milliseconds, with a standard deviation of 30 microseconds, for recovery. Protecting `ssh` only requires erasing the session key. Recovering the session key requires two round-trips to the token to recover the outgoing and incoming session keys. An alternate implementation could recover both session keys simultaneously, but the cost is already small enough.

6.4 Mozilla

The only overhead to Mozilla's normal operation is the use of stored password data and cookies. Each of these are encrypted and decrypted on each use. Passwords are already SDR-encrypted and decrypted by Mozilla; our version does not add any overhead to this. To benchmark the cost of cookies, we loaded three popular pages and measured the total overhead of encryption and decryption. To put these costs in context, we also report the fastest load time we observed for each page. The cookie store was cleared between each trial and the mean and standard deviation are reported in Figure 10. For theses pages, the additional overhead of encrypting and decrypting cookie data is small enough to be masked by page loading times.

	Overhead,sec	Load Time,sec
CNN	0.010 (.004)	3.1
Ebay	0.035 (.001)	1.7
ESPN	0.134 (.004)	3.8

Figure 10. Mozilla Cookie Overhead

We also measured the amount of time required to protect and restore Mozilla when the user leaves. To measure this we connected to two secure sites, a bank and our own department's secure web server. We disconnected the token and measure the time to safety, then reconnected the token and measured the recovery time. The results for each component are shown in Figure 11. We also report the amount of data in the memory cache, and the amount of data consumed by SSL keys.

	Protect, sec	Restore, sec
Memory Cache (518 kB)	0.222 (0.002)	0.222 (0.004)
SSL Keys (788 bytes)	0.003 (0.000)	0.074 (0.006)
SDR (16 byte key)	N/A	0.066 (0.005)

Figure 11. Mozilla Protection and Recovery

Recall that the contents of the memory cache and the SSL keys are encrypted when the user leaves. The memory cache, stored passwords, and cookies depend on SDR for encryption support, so SDR's key must be flushed on departure and recovered on return. Flushing the key takes a negligible amount of time. SSL uses its own key for protecting the SSL keys, and must recover it when the user returns. The total time to secure and restore Mozilla is less than four tenths of a second. Compared to the amount of time between a user entering range and resuming work, this cost will not be visible.

7 Related Work

Several efforts have used proximity-based hardware tokens to detect the presence of an authorized user. Landwehr [15] proposes disabling hardware access to the keyboard and mouse of a machine when the trusted user is away. A commercial alternative, XyLoc [27], has a software-based guard on the protected machine that refuses access when the token is absent. These systems approximate Transient Authentication, but do not adhere to its first principle. The capability to act in these systems does not reside on the token; the token is merely advisory. Since the computing system is still capable of carrying out a sensitive operation, it could be forced to do so. Sensitive operations may be relegated to a secure coprocessor [14], rendering these physical attacks more difficult.

Rather than use hardware tokens, one could instead use biometrics. However, biometric authentication schemes intrude on users in two ways. The first is the false-negative rate: the chance of rejecting a valid user [22]. For face recognition, this ranges between 10% and 40%, depending on the amount of time between training and using the recognition system. For fingerprints, the false-negative rate can be as high as 44%, depending on the subject. The second intrusion stems from physical constraints. For example, a user must touch a special reader to validate his fingerprint. Such burdens encourage users to disable or work around biometric protection. A notable exception is iris recognition. It can have a low false-negative rate, and can be performed unobtrusively [20]. However, doing so requires three cameras—an expensive and bulky proposition for a laptop.

For Transient Authentication to succeed, a computing device must *forget* sensitive information, typically through encryption. Thereafter, only the token can provide the key to recover this information. Such techniques have also been applied to revocable backups [2] and secure execution of batch jobs [24], and are sometimes referred to as non-monotonic protocols [25]. It can be difficult to completely erase previously stored values, whether in memory or on disk [12]. However, given a small amount of easily erasable media one can solve this problem for a much larger, more persistent store [9].

ZIA, a cryptographic file system, uses Transient Authentication for file data protection [6]. ZIA imposes overheads of less than 10% for representative workloads, and imposes no new usability burdens. Unfortunately, ZIA does not protect data once an application has read it. Application data that is paged out can be protected [23], leaving only in-memory state vulnerable to attack.

8 Conclusion

Mobile devices are susceptible to loss or theft, leaving the state of running applications vulnerable to data exposure. Current methods of authentication do not solve this problem since authentication is both infrequent and persistent. As a solution to this problem, we propose Transient Authentication, which allows a system to constantly reaffirm the capability to read sensitive data from memory, while giving the user no reason to turn protections off.

In this paper, we have demonstrated two protection methods that use Transient Authentication support. One mechanism is transparent, operating without application modification. The second is an API that gives greater flexibility to application designers in dealing with authentication. The evaluation of these two techniques shows that transparent protection can both secure and recover the entire physical memory of the machine within 10 seconds and that the API can be used to secure a complex application within four tenths of a second.

References

[1] D. Atkins, W. Stallings, and P. Zimmermann. Pgp message exchange formats. RFC 1991, August 1996.

[2] D. Boneh and R. J. Lipton. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium*, pages 91–96, San Jose, CA, July 1996.

[3] S. Brands and D. Chaum. Distance-bounding protocols. In *Proceedings of EUROCRYPT '93*, Lecture Notes in Computer Science, no. 765, pages 344–359. Springer-Verlag, 1993.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[5] F. Chabaud. Linux software suspend. http://sourceforge.net/projects/swsusp.

[6] Reference withheld to preserve anonymity.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer overflow attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–77, San Antonio, TX, January 1998.

[8] J. Daemen and V. Rijmen. AES proposal: Rijndael. Advanced Encryption Standard Submission, 2nd version, March 1999.

[9] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, M. Jakobsson, C. Meinel, and S. Tison. How to forget a secret. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects in Computer Science*, pages 500–509, Trier, Germany, March 1999.

[10] W. Diffie, P. van Oorschot, and M. Wiener. *Design Codes and Cryptograhpy*. Kluwer Academic Publishers, 1992.

[11] A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. Internet Draft, March 1996.

[12] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–89, San Jose, CA, July 1996.

[13] Y. Hu, A. Perrig, and D. B. Johnson. Wormhole detection in wireless ad hoc networks. Technical report, Rice University Department of Computer Science, June 2002.

[14] IBM. IBM client security solutions. White Paper, November 1999.

[15] 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.

[16] B. A. Miller and C. Bisdikian. *Bluetooth Revealed*. Prentice Hall, Upper Saddle River, NJ, 2001.

[17] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[18] 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.

[19] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–9, December 1978.

[20] 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.

[21] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.

[22] 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.

[23] N. Provos. Encrypting virtual memory. In *Proceedings of the Ninth USENIX Security Symposium*, pages 35–44, Denver, CO, August 2000.

[24] A. D. Rubin and P. Honeyman. Long running jobs in an authenticated environment. In *Proceedings of the 4th USENIX Security Symposium*, pages 19–28, Santa Clara, CA, October 1993.

[25] A. D. Rubin and P. Honeyman. Nonmonotonic cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop*, pages 100–116, Franconia, NH, June 1994.

[26] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*, chapter 5: CPU Scheduling. John Wiley & Sons, Fifth edition, 1999.

[27] Ensure Technologies. http://www.ensuretech.com/.

[28] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the First USENIX Workship of Electronic Commerce*, pages 155–70, New York, NY, July 1995.

[29] T. Ylonen. SSH—Secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.