Secure Distribution of Confidential Information via Self-Destructing Data

JASON CROFT
Boston College
Computer Science Department
Chestnut Hill, MA, 02467
USA
croftj@bc.edu

ROBERT SIGNORILE
Boston College
Computer Science Department
21 Campanella Way, 572, Chestnut Hill, MA, 02467
USA
signoril@bc.edu

Abstract: Control and ownership of data is difficult in any environment and with the increase in electronic data and records, the need to maintain ownership and control redistribution of data is becoming increasingly important. We propose a first-level protection against unauthorized redistribution using a method of self-destructing, one-time-use data. Transmitted data is encrypted, encapsulated within an executable, and authenticated to a single user and machine. Once accessed, measures are taken to ensure it cannot be used outside the executable (e.g., displayed within a non-selectable, non-editable window) and that the executable cannot be easily decompiled. After a single use, data is destroyed through a method of in-memory compilation of a new executable, which overwrites the original during runtime. In addition, a time-to-live (TTL) is integrated into the executable to provide an additional layer of security so that the data is only accessible within a defined time period. The executable is self-sufficient–it requires no network connection, communication with a central authority, or communication with the sender to authenticate the data since all authentication is integrated into the executable. This provides universal, environment-neutral protection of the data within any type of transfer, whether via server-client, peer-to-peer (P2P), or through external storage devices.

Key–Words: Data Security, Computer Networks, Data Retrieval

1 Introduction

Self-destructing data may be the fiction of spy movies like *Mission Impossible*, but it has many applications in controlling distribution of data in real world situations. For example, it can be used to securely distribute confidential data, with the assurance that no unauthorized distribution of data will follow. This can be an often overlooked security weakness because of the difficulty in controlling it. Secure data transfer is not merely enough for data security–secure transfer and secure distribution are needed. If Alice wishes to send a file to Bob, she can encrypt the data to ensure only Bob can decrypt it. However, once decrypted, Bob can redistribute it to an unauthorized user Eve, or it could be susceptible to a data-comprising attack that may result in Eve obtaining the data. In either case, the confidentiality of the data is lost.

To mediate this problem, we propose a method of first-level protection against unauthorized distribution that is built on self-destructing data. The self-destruction nature of the data is built on in-memory compilation and data encapsulation within executables. When a user distributes data to another user, the data is authenticated to the receiving user and then wrapped within an executable Java class file. Though this method has some restrictions, we propose methods to mediate any potential problems and strengthen its security with minimal impact on usability. Our current implementation works for data such as text, HTML, or PDFs, and we leave other types (*e.g.*, videos, music) to future work. It also relies on a proprietary method to display data (*i.e.*, a "data viewer") and control the destruction of data.

Additionally, this model is completely independent of any type of data-transfer environment and provides universal and pervasive protection of confidential data. That is, data can be protected within any type of network architecture, such as server-client or peer-to-peer (P2P), or even in the absense of a network, such as data transfer via external storage. With authentication built into the executable rather than requiring a central authority or peers to negotiate authentication, the data is secure across any type of data transfer.

2 Applications

2.1 HIPAA

Secure distribution has many applications in patient confidentiality under the Health Insurance Portability and Accountability Act (HIPAA) [2]. Especially important are the Administrative Simplification provisions and the "Standards for Privacy of Individually Indentifiable Health Information" (Privacy Rule), which set standards for electronic

health care transactions and address the security and privacy of health data. With more and more hospitals using electronic records, the need for secure distribution of this information is becoming increasingly important. Annas [12] clarifies the three basic rights a patient should have in keeping medical information private:

- Physicians have an obligation to keep medical information secret
- Patients are unlikely to disclose intimate details unless their physician is trusted to keep the information secret
- No entity should have access to records without a patient's authorization

Baumer et al. [14] discuss some of the security implications of electronic records and note, in particular, the issue of unauthorized secondary usage of patient data. In addition, their survey found a significant concern among healthcare workers about "inappropriate and unauthorized access to medical records" and noted that "healthcare workers take most seriously unauthorized secondary use of medical information"

2.2 Classified Information

Any type of sensitive information, particularly within the government sector, could benefit from self-destructing data. Strong authentication and secured networks can prevent data loss from the outside-in, but without a method to prevent data loss internally, the overall security is weakened. Any data to be electronically transferred can be authenticated to the users meeting the security clearance, and both prevents unauthorized use of the data as well as unauthorized distribution. In essence, this level of protection is as important as the need for encryption in maintaining security of the data.

2.3 Corporate Information

Similarly, sensitive corporate information, such as trade secrets, strategic plans, new product information, or merger information can be secure using our method. This could potentially prevent any type of unauthorized disclosure or leaked information. Similarly, the same can apply to copyrighted material as a form of digital rights management (DRM) to prevent unauthorized distribution. With a secure method of distribution, nearly any type of "endpoint data loss" can be prevented. However, given the design of our model and the environment-neutral security it provides, it is not limited to securing data on a corporate network but also from any type of storage device.

2.4 Pairwise Trust

In our design, we assume some pairwise trust between two *trusted* users. That is, we assume a receiving user will not attempt to maliciously redistribute data. This case applies to corporate or government information in which the receiving individual has the appropriate clearance, *i.e.*, s/he is trusted to use the information properly. The need for secure distribution in this case is to prevent any type of accidental loss of data, whether through lost hardware, a malicious third party, or even accidental redistribution. This gives the receiver some assurance that the executable with the encapsulated data does not contain a virus or malware.

Nevertheless, our design protects against even a malicious receiver. Whether this receiver is trusted by the sender but acts maliciously, or is untrusted to the sender but is still requesting data, the data can still be protected from redistribution. The need for this type of trusted-but-insecure security is evident especially in HIPAA-related confidentiality.

3 Related Work

We could find very little research on self-destructing data, with most of the emphasis on self-destructing email. The most concrete example was in a patent for self-destructing email [27]. In it, the authors describe this design as "automatically [destroying] documents or email messages at a predetermined time by attaching a 'virus' to the document or email."' However, this differs significantly from our design, as we focused on less intrusive methods to avoid issues with anti-virus or anti-malware software.

In addition, SafeMessage attempts to control document distribution using recipient and status verification, negotiated single-session encryption, and limited persistence [7]. Confidentiality of the email is ensured by encrypting emails to the intended recipient and storing them on a single SafeMessage server. Upon retrieval, the message can be destroyed. VaporStream [9] achieves a similar goal by separating a messages header and body, storing it in a temporary buffer, and then removing it once the recipient has retrieved the message. Several other services exists [3, 8, 10], but all essentially rely on propriety software on a dedicated server.

In regards to HIPAA compliance, several services have been examined to enforce privacy and confidentiality of patient information. Cao's approach [15] uses a digital envelope concept that provides image integrity and security assurance. This envelope, as well as the digital signature, is then embedded in the background of the image as an invisible watermark. A picture archiving and communication system (PACS) is also used. Yee and Trockman's SafeByte [28] is an electronic personal health record that uses executable software and data files for distributing information, while Clarke et al. [17] design a Communication Virtual Machine (CMV) to enforce privacy and security requirements.

Though not focusing on self-destructing data, other work has taken different approaches to controlling data dis-

tribution by implementing "persistent access control" [20, 13] or using trust-based access control [26, 19, 11]. Microsoft Office's Information Rights Management (IRM) [4] is similar, aimed at preventing an unauthorized user from forwarding, copying, modifying printing, or pasting content. It also supports file expiration–similar to self-destructing data. However, IRM is implemented as a Web Service for Microsoft enabled products. While these have their strengths, the dependency on trusted hardware can be restraining, and as such, we sought to propose another solution. Examples of enterprise software aimed at protecting against "endpoint-data loss" include NextLab's Enterprise DLP Data Protection suite [1], Symantec's Data Loss Protection [25], and RSA's Data Loss Prevention Endpoint [6].

4 Architecture

Our model for secure distribution relies on self-destructing data, which we achieve through data encapsulation. The confidential information is stored within an executable Java class file that renders the data once per authenticated user. The sender/owner authenticates the data to a single user and machine to control access to the data. All functionality to authenticate the receiver is built into the executable, so no further communication with the sender is required after distribution. After the first use of the data, the data is destroyed using a method of in-memory compilation that overwrites the class file. This combination of authentication and self-destruction is the key to establishing a first-level protection of distribution of data.

The goal of this executable is to prevent the receiving user from:

- Using the data on different machine or using a different username than authenticated
- Redistributing the data to a non-authenticated user/machine

4.1 Data Authentication

All transferred data is encapsulated within an executable Java class file, such that only the data owner(s) store it in the original format. Within the executable, the data is stored as a base64 string. Obfuscation is used to impede reverse engineering or disassembly. As discussed later, we used ProGaurd [5] in our proof of concept for obfuscation. If these authentication factors cannot be checked at runtime due to security restrictions on the Java Virtual Machine (JVM), the authentication is assumed to have failed and the data is destroyed. This two-factor authentication limits one user on one machine to view the data. Only after authentication of the user's MAC address and username succeeds will the data be decrypted and made viewable. A hash of the authentication factors is used to decrypt the data, which is rendered on-screen in a non-editable, non-selectable region to prevent redistribution through a simple copy and paste. If the authentication should fail, the data is not decrypted and the class file is overwritten using in-memory compilation. The new executable contains none of the confidential data, so after compilation the data is destroyed.

4.2 Self-Destructing Data

Our method of self-destructing data is less intrusive than the virus-appended approach in [27]. We use in-memory compilation to overwrite class files. New source code, stored as an encrypted string within the executable, is decrypted then compiled in-memory. The new source contains none of the original data, but only the functionality to notify the sender of subsequent policy violations. Given write privileges on the class file, the Java Virtual Machine (JVM) will allow this file to be overwritten. This process is shown in Fig. 1.

In-memory compilation is vital for security of the application. If we store the code as a string, decrypt it, then attempt to compile it from disk, a malicious user can potentially use the knowledge of the source code for exploitation. Even attempting deletion of the source file after compilation presents a security risk as it is nonetheless stored on disk at some point. In encapsulating the data into the executable, the goal is to provide the receiver with the minimum knowledge needed to consume the data. Using in-memory compilation, no additional data must be stored outside of the executable. External information is far more susceptible to modification by a malicious user than internal data. Data stored information in registry keys, for example, is easier to modify than variables stored within the class file.

Screenshots of the self-destructing data and in-memory compilation are shown in Fig. 2. On the right hand side, the executable is compiled, authenticating the file "Test.txt" to the current user and machine and then executed. After the initial compilation, the executable (containing the encapsulated data is 59KB. After successful authentication, the data is rendered on-screen. On the left hand side, the data viewer is closed and the file size of the executable is shown as 5KB. After the in-memory compilation during the data render, the data is destroyed, as evident in the class file reducing from 59KB to 5KB. When another execution is attempted, the user is notified that no other views are allowed.

4.3 Current Dependencies and Requirements

The data viewer requires several dependencies for full functionality. Any unmet dependency will result in reduced security, so we require all be met for the viewer to execute. First, the client machine requires the current version of the

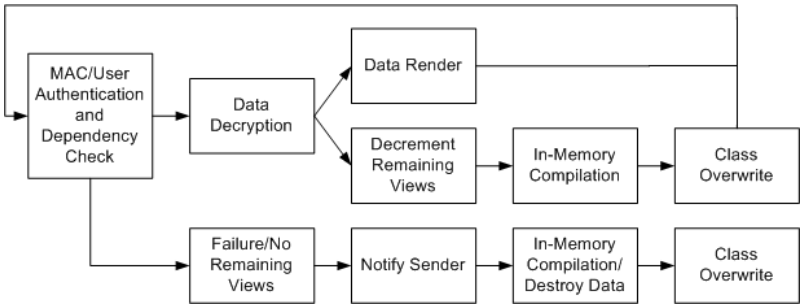

Figure 1: Data viewer process

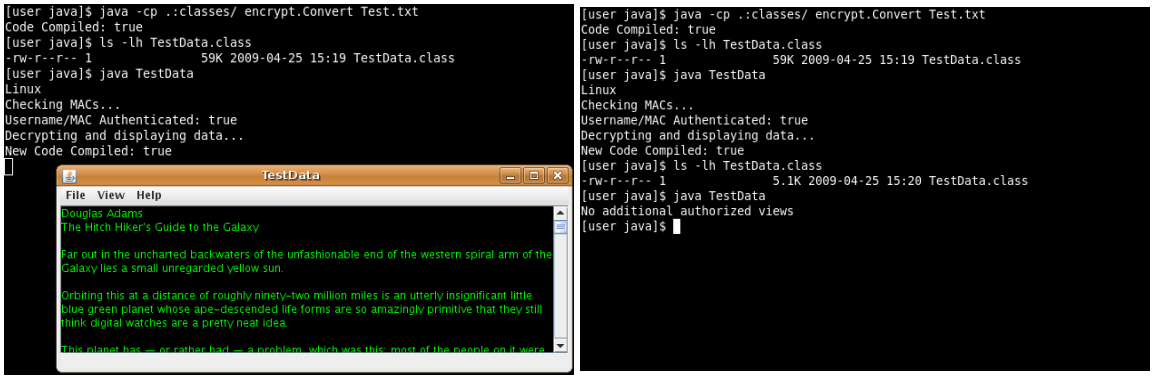

Figure 2: Screenshots of the self-destructing data/data viewer

Java Development Kit (JDK), version 6. This release supports the `javax.tools` package, which is needed for the in-memory compilation.

Secondly, write privileges are required on the received class file. In our test cases, the received class files are given write privileges for the receiving user. Only after intentionally changing the file permissions was write access revoked. We assume a malicious user would use an attack of this sort to prevent the executable from overwriting itself and delay data destruction. In this case, the user has the potential to attempt reverse engineering of the class file to obtain the data since it cannot be overwritten.

For the in-memory compilation, few restrictions exist. First, class files have a size limitation. The value of the file must be less than 64KB [18], as per the JVM class file specification. Data files larger than this can be split into multiple strings, stored in separate class files, and then coalesced at runtime.

4.4 Security Considerations

The goal of our design is to provide some level of ownership over data once an owner has distributed it onto the network. As such, the decrypted data will be in memory for the duration the viewer is running. Since data is neither selectable nor editable, a user cannot copy/paste the data, but could use a core dump to retrieve the current on-screen data. A screenshot or screen-capturing software could also be used to retrieve the data. For large amounts of data, this is a tedious task. Furthermore, since we assume some initial pairwise trust between the sender and receiver, the sender should be able to assume a receiver would not attempt to redistribute the data in this manner.

Reverse engineering of the class file is another consideration. We used ProGuard to obfuscate the class files containing the data, making reverse engineering a more difficult task. More advanced techniques include [16, 23]. The class files containing the data could be encrypted during the transaction, then decrypted at runtime by the class loader. Though this is not a fully robust and sound solution, it nevertheless provides an additional level of security a malicious peer must overcome.

If an attack potentially knows which authentication factors are used to authenticate the receiver, the authentication policy can be circumvented. One possible solution is random authentication factors. Two random factors of several possible ones could be used, such as the username, MAC address, or CPUID. While a malicious user will have no knowledge of which system properties are being used, the soundness of the authentication is decreased if one of the available factors in not incorporated.

One potential attack against the limited-use is filesys-

tem or virtual machine (VM) snapshots. However, we consider circumventing this policy to be only a minor consideration, as the goal of our work is in preventing unauthorized redistribution. Multiple VM snapshots will still not allow a user to distribute the secured data to another user.

5 Proof-of-Concept

To demonstrate our proposal, we have implemented a proof-of-concept content distribution system that utilizes this self-destructing data model. The content distribution system allows users securely transfer data across a network, but prevents this data from being redistributed. We utilized a P2P overlay for data discovery, allowing users to connect using a application similar to P2P file sharing software like Gnutella, but used secure, unicast connections for data transfer.

To prevent distribution of malicious files, such as viruses or malware concealed as confidential data, all encapsulation of data is done by the distribution system. Sending a file is a simple as selecting a file and user to send–the encryption, authentication, and transmission are carried out by our application. A malicious user could attempt to manually send a secure file to another user via email, but like any file received over email or from some non-secure source, the receiving user should take caution. With our design, the automatic generation of the executable gives some assurance that it is not malicious. For the limited-use policy, we enforced a one-time-use limitation.

5.1 Network

While our idea is applicable to any type of environment for data transfer (*e.g.*, client-server network, P2P network, or transfer via external storage), we chose to implement it using a P2P network to demonstrate several important ideas. First, unlike the endpoint data loss protection systems, there is no need for a central authority. This allows for use and protection of the data within a user's network, outside the network, or without any connection. In addition, networks with high resource relevance to participants benefit from the strengths of P2P networks [22].

Our P2P network acts as an overlay on an enterprise network that requires authentication to join, such as an internal academic or corporate network. This establishes some initial trust between users–provided that the network authentication is sound–since a user's information cannot be spoofed. Though not implemented in our proof-of-concept, another proposed method to prevent spoofing is to leverage DHCP information. Both the LDAP service and DHCP information can be used to strengthen the authentication required for the data.

5.2 Data Transfer

Data discovery is done manually; a user can select the data they wish to share, and this information is propagated on the network. However, since we would anticipate large numbers of users and files, in a more advanced proof-of-concept this would be done using a distributed hash table, such as CAN [21] or Chord [24]. When a user needs some data, s/he requests it from the sender. Then sender has the option to accept or deny the request. If accepted, a separate unicast connection is established in addition to the already existing multicast connection between all peers on the network. The content distribution then uses the receiver's information (MAC address and username are discovered at runtime and then relayed to the sender at the data request) to compile the executable and send it to the receiving peer. This allows for minimal work on the sender side to secure the confidential data.

5.3 Trust

In addition, we chose to incorporate some trust information about a receiver's use of the data. Despite the assumed initial pairwise trust, peers could assess another peer's trustworthiness without any prior first-hand history. We defined misuse of the data to be:

- Attempted additional usage of the data
- Attempted redistribution of the data
- Failed authentication

Misuse would lower a peer's trustworthiness while proper usage would increase it. Attempted redistribution lowers the trust of both the intended receiver and the third user receiving the unauthorized data. "Attempted" implies the user executed the encapsulated data to try to view the data, but failed because of the security mechanism integrated into the executable. Thus, a user trying to view the data a second time would merely encounter a warning that the number of alloted views has been surpassed. To propagate data usage information, we built a feedback mechanism into the executable. If the receiver is connected to a network during usage of the data, the executable will attempt to reconnect to the content distribution system's P2P network and relay the usage information to the sender. We chose not to enforce network access as a requirement since it is severely limiting, so we incorporated an additional timeout for the user's reappearance on the network under the assumption that a malicious user would remain disconnected from the network until the data could be extracted from the executable.

6 Conclusions and Future Work

We have implemented a secure method to transfer data using self-destructing data. This model is data-transfer

neutral, that is, it protects data throughout any type of electronic data transfer, regardless of network architecture. Data can be securely distributed in server-client networks, P2P networks, or via external storage. Due to our use of Java to create executables, data usage is also platform independent and can be transferred between any systems meeting the requirements described earlier. We have demonstrated the ability to restrict access of data to authorized users and limit the time it is available as an additional method of security.

While this current implementation may succeed in preventing unauthorized data distribution, it does so with a read-only limitation. Allowing outright write-access is difficult due to a method of storing these changes locally. One idea we propose is annotations and requests for deletion. If Alice sends a file to Bob, Alice would have the ability to add annotations to the data and mark areas for deletion. This must be done with care as the data viewer must still be non-selectable and non-editable to prevent copy-and-pasting of the data outside the viewer. We would like to increase the usability of this concept from text-based data to other types of data and media. Our current design relies on a proprietary method to view the data, which imposes some limitations on its use. First, we hope to extend our idea to other data formats. Once this is done, we would like to determine a method of using this model without the need for proprietary software. That is, data can be viewed using the default application, but still retain its self-destructing security.

References:

[1] Endpoint Data Loss Prevention. http://www.nextlabs.com/html/?q=endpoint-data-loss-prevention.

[2] Health information privacy. http://www.hhs.gov/ocr/privacy/index.html.

[3] Kicknotes self-destructing email. http://www.kicknotes.com/.

[4] Microsoft office information rights management(irm). http://www.microsoft.com/ windowsserver2003/techinfo/ overview/rmenterprisewp.mspx.

[5] Proguard. http://proguard.sourceforge.net/.

[6] RSA Data Loss Prevention (DLP) Endpoint. http://www.rsa.com/node.aspx?id=3429.

[7] Safemessage. http://www.safemessage.com/.

[8] Self destructing email. http://www.self-destructing-email.com/.

[9] Vaporstream. https://www.vaporstream.com/.

[10] Zmail basic. https://zsentry.com/zmail/.

[11] W.J. Adams and IV Davis, N.J. Toward a decentralized trust-based access control system for dynamic collaboration. pages 317–324, June 2005.

[12] George J. Annas. Hipaa regulations - a new era of medical-record privay. *New England Journal of Medicine*, 384:1486–1490, April 10 2000.

[13] Alapan Arnab and Andrew Hutchison. Persistent access control: a formal model for DRM. pages 41–53, 2007.

[14] David Baumer, Julia Brande Earp, and Fay Cobb Payton. Privacy of medical records: It implications of hipaa. *SIGCAS Comput. Soc.*, 30(4):40–47, 2000.

[15] F. Cao. Medical image security in a hipaa mandated pacs environment. *Computerized Medial Imaging and Graphics*, 27(2-3):185–196, June 2003.

[16] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *J. Syst. Softw.*, 71(1-2):1–10, 2004.

[17] Vagelis Hristidis, Peter J. Clarke, Nagarajan Prabakar, Yi Deng, Jeffrey A. White, and Redmond P. Burke. A flexible approach for electronic medical records exchange. In *HIKM '06: Proceedings of the international workshop on Healthcare information and knowledge management*, pages 33–40, New York, NY, USA, 2006. ACM.

[18] Tim Lindholm and Frank Yellin. *Java(TM) Virtual Machine Specification, Second Edition*. Prentice Hall, 1999.

[19] Xiaoning Ma, Zhiyong Feng, Chao Xu, and Jiafang Wang. A trust-based access control with feedback. pages 510–514, May 2008.

[20] Paul B. Schneck. "Persistent Access Control to Prevent Piracy of Digital Information". *Proceedings of the IEEE*, 06/1999.

[21] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[22] Mema Roussopoulos, Mary Baker, David S. H. Rosenthal, Tj Giuli, and Jeff Mogul. 2 p2p or not 2 p2p. In *In IPTPS 04*, pages 33–43. Springer, 2004.

[23] Yusuke Sakabe, Masakuzu Soshi, and Atsuko Miyaji. Java obfuscation approaches to construct tamper-resistent object-oriented programs. *IPSJ Digital Courier*, 1:349–361, 2005.

[24] Ion Stoica, Robert Morris, David Karger, Frans M. Kaashoek, and Hari. Chord: A scalable peer-to-peer lookup service for internet applications, 2001.

[25] Symantec. Data loss protection. http://www.symantec.com/business/data-loss-prevention.

[26] Huu Tran, M. Hitchens, V. Varadharajan, and P. Watters. A trust based access control framework for p2p file-sharing systems. pages 302c–302c, Jan. 2005.

[27] Howard R. Udell, Cary S. Kappel, William Ries, Stuart D. Baker, and Greg M. Sherman. Self-destructing document and e-mail messaging system". US Patent Number 7191219, April 12, 2002.

[28] Wai Gen Yee and Brett Trockman. Bridging a gap in the proposed personal health record. In *HIKM '06: Proceedings of the international workshop on Healthcare information and knowledge management*, pages 49–56, New York, NY, USA, 2006. ACM.