Secure Information Sharing and Processing (SISAP) Technology

Cadet Daniel Eichman
US Air Force Academy
C16Daniel.Eichman@usafa.edu

Abstract. This is a report of a 7 week research program conducted at MIT Lincoln Laboratory. Secure Information Sharing and Processing (SISAP) was designed to tackle the "trusted user" problem. The "trusted user" problem begins with a user who is deemed "trustworthy" and thus given access to sensitive information; at a later point in time, this user becomes compromised and will now have unlimited access to sensitive information. SISAP creates the next generation office environment in which user permissions can be extremely fine-tuned. The vision of SISAP is that SISAP users may only access data at certain work stations or a predefined set of hardware devices.

Keywords: Trusted User, Encryption Enforced Policy

1 Introduction

With recent insider information leaks such as the infamous Edward Snowden case, system administrators have begun to question the sensibility of placing complete trust on their system users. Traditionally, in systems, the "good guys" had full access to data while the "bad guys" had none. The problem with this is that there is no way of telling who is a good guy and who is not. This is where SISAP becomes extremely useful. Imagine a policy that allows only certain users to access data while physically at the office or on certain machines. Thus even if the bad guys did manage to copy and steal data, they would have no way of processing it elsewhere.

1.1 Goals

The goals of the project are to (1) create an encryption scheme that would reduce trust placed in the user, (2) enforce policy through encryption, (3) allow certain subsets of users to access specific permissions, (4) not place full confidence in the server or user, and (5) be transparent to the user.

1.2 Future Office Space

SISAP is designed so that data is tied to both, users and hardware, while in a traditional system the data is only tied to users. The current situation, described in the Figure 1, shows that once a user has access to a file, there is no way to continue to limit his ability to process it. While in SISAP, even if you have access to the file you may only be able to display it on a certain monitor or print at a certain printer. To achieve this, everything in the system has a set of public and private keys, creating a

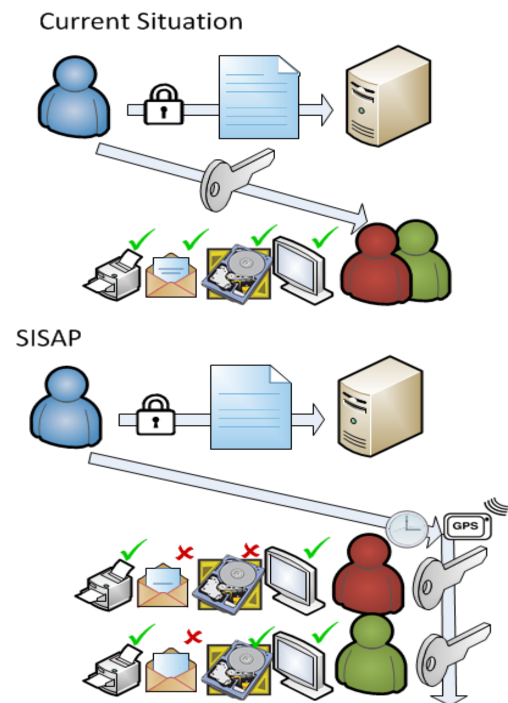

Figure 1

Distribution A: Public Release. This work was sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002 Opinions, interpretations, recommendations and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

giant PKI database. These keys allow each file to have specific encrypted permissions, which grant users access. Ultimately, SISAP is designed to create the next generation office space, where every output device and user is part of the encryption scheme. Current output devices will have to be modified to be incorporated into the system, but this cost will allow fine-tuned user permission and greater control over data.

1.3 Initial Design Failures

This problem proved to be more complicated than initially thought. Many of the initial designs had major security flaws or failed to properly incorporate desirable features. A few features that were left out in the final design were attribute-based encryption and secure logging. Through design evolution, I found that the only "trusted" elements in the system were the hardware devices themselves. This forces the trust to be distributed to the entire system. One of the benefits of this distribution of trust is that if a device is compromised, less information will be leaked than if the compromise occurred with a centralized trusted device.

1.4 Physical Limitations

While SISAP aims to prevent data from being leaked, there are some physical limitations. The first and most obvious is screen capture tools. If a user has the permission to open a file it is difficult to stop him from using a screen capturing tool and almost impossible to stop him from using a camera. Similarly, if a user has the ability to print a document there is no telling what that user may do after the document is printed.

2 Creating a File

The first step when adding a content file to SISAP is to encrypt it with a symmetric key. This key is called the content key. Anyone with access to the content key can decrypt the content of the file, therefore this key needs to be protected. Figure 2 below shows the content file (MyFile.txt) being encrypted with a symmetrical content key (red).

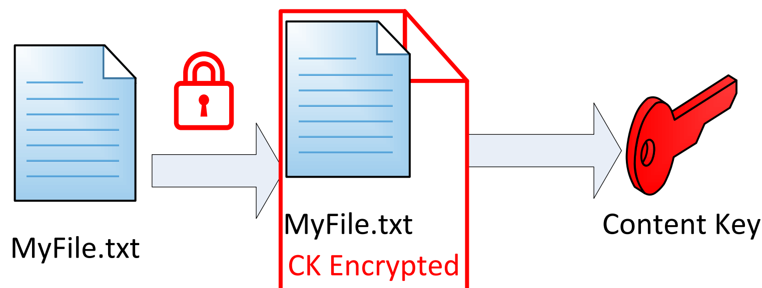

Figure 2

The next step is to protect the content key. In our system, neither the server nor the users are trusted; this creates an interesting predicament, as we do not know how to protect the content key. This leaves us with trusting our only option: trusting the hardware devices. This is actually an advantage as the policy is enforced at the devices not at the server. So even if one device is compromised, only data allowed at that device will be compromised. To protect the content key, we wrap it with a specific set of hardware devices' public keys, so that only hardware devices are able to decrypt it.

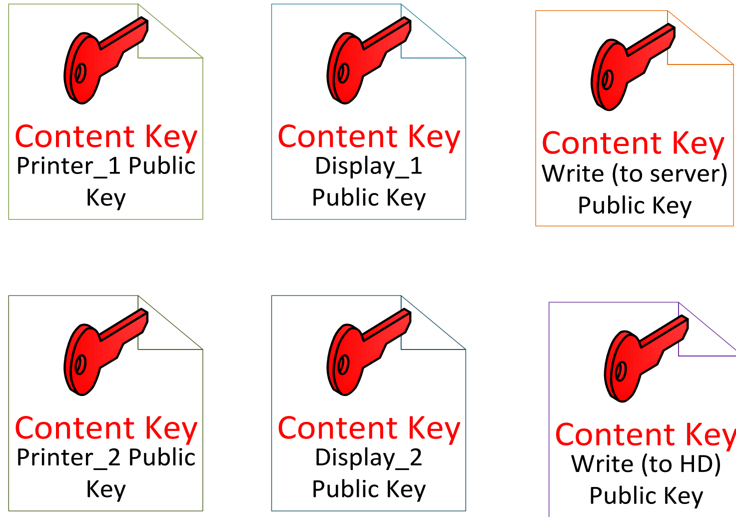

Figure 3

In every system some users must be able to process certain files. In SISAP we grant access to files by wrapping the content key (already wrapped with the hardware keys) with the user's public key. To access this permission, the user must decrypt the key, verifying that the user has permission to access the file. This allows SISAP to enforce the policy with encryption. If the user cannot decrypt the key, the user will obviously not be able to use the file. This eliminates the need for the server or device to make a Boolean decision, which should make the system more secure.

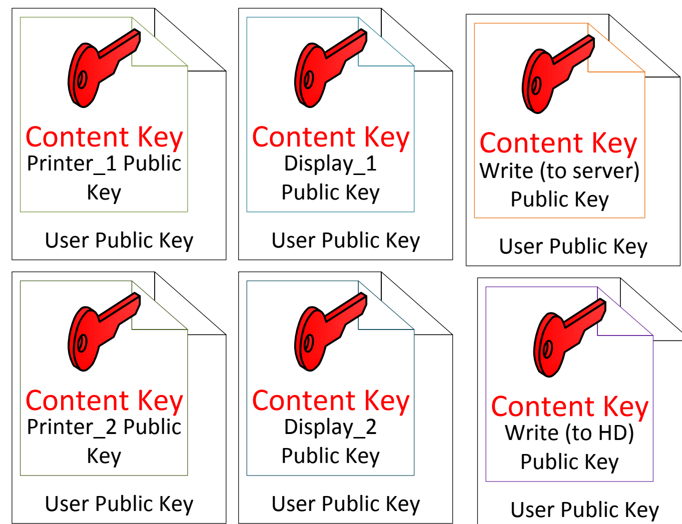

Figure 4

These permissions will be stored together with the file on the server. The server will be in charge of distributing permissions and keys; however it will not be in charge of enforcing policy as that will be done by encryption.

3. Accessing a File

3.1 Commutative and Cascadeable Cryptography

SISAP relies heavily on commutative or cascadeable cryptography. This is when the order of encryption can be different than the order of decryption while remaining semantically secure[1]. The beauty of this system is that cypher text, $C = e_2 e_1(m)$, can be decrypted into two different orders such that $m = d_2 d_1(C)$ and $m = d_1 d_2(C)$. An example of a public-key cryptosystem that is commutative is the ElGamal encryption scheme originally conceived by Taher Elgamal, and based on the Diffie-Hellman key exchange protocol. The commutative property ElGamal encryption scheme is quintessential for making SISAP work.

3.2 Proving identity

In the earlier stages of the design, identity was proven by showing knowledge of some known fixed secret. This secret was encrypted with the approved user's public key, such that if you were an approved user you would be able to prove your identity by decrypting the secret. Unfortunately, our threat model does not trust the users and assumes they could potentially be malicious. What if the user stores the secret after she decrypts it, or even worse, what if she gives that secret to another user so he now has access as well? Clearly this model has some short comings, and this is where commutative cryptography can be applied. In SISAP, the content key will be wrapped with the hardware keys, which will be equivalent to the known secret mentioned earlier. This secret, when decrypted at the hardware, can unlock the content key, which gives access to the file. This key warp will be stored and encrypted using the user's public key. When accessing the key, a set of session keys will be created. Then the stored key will be encrypted with a session key. This will result in $C = e_{Session} e_{user}(key)$. Because ElGamal is commutative, the key can be decrypted in any order $key = d_{Session} d_{user}(C)$. In SISAP, the user will decrypt first to prove the user's identity while simultaneously revealing nothing to the user himself. Then the hardware device will decrypt next, revealing the key, which can then be used. Since the session will be renewed each time the user accesses this permission, the user will always receive a different cypher text $C = e_{Session} e_{user}(key)$and decrypt to a different cypher text $C = e_{Session}(key)$. This ensures that the user has no knowledge of the key, which means they cannot leak any information as the session will expire and their cypher text becomes useless.

---

[1] Weis, Stephen A. *New Foundations for Efficient Authentication, Commutative Cryptography, and Private Disjointness Testing*. Thesis. Massachusetts Institute of Technology, 2006. Cambridge: Massachusetts Institute of Technology, 2006. Print.

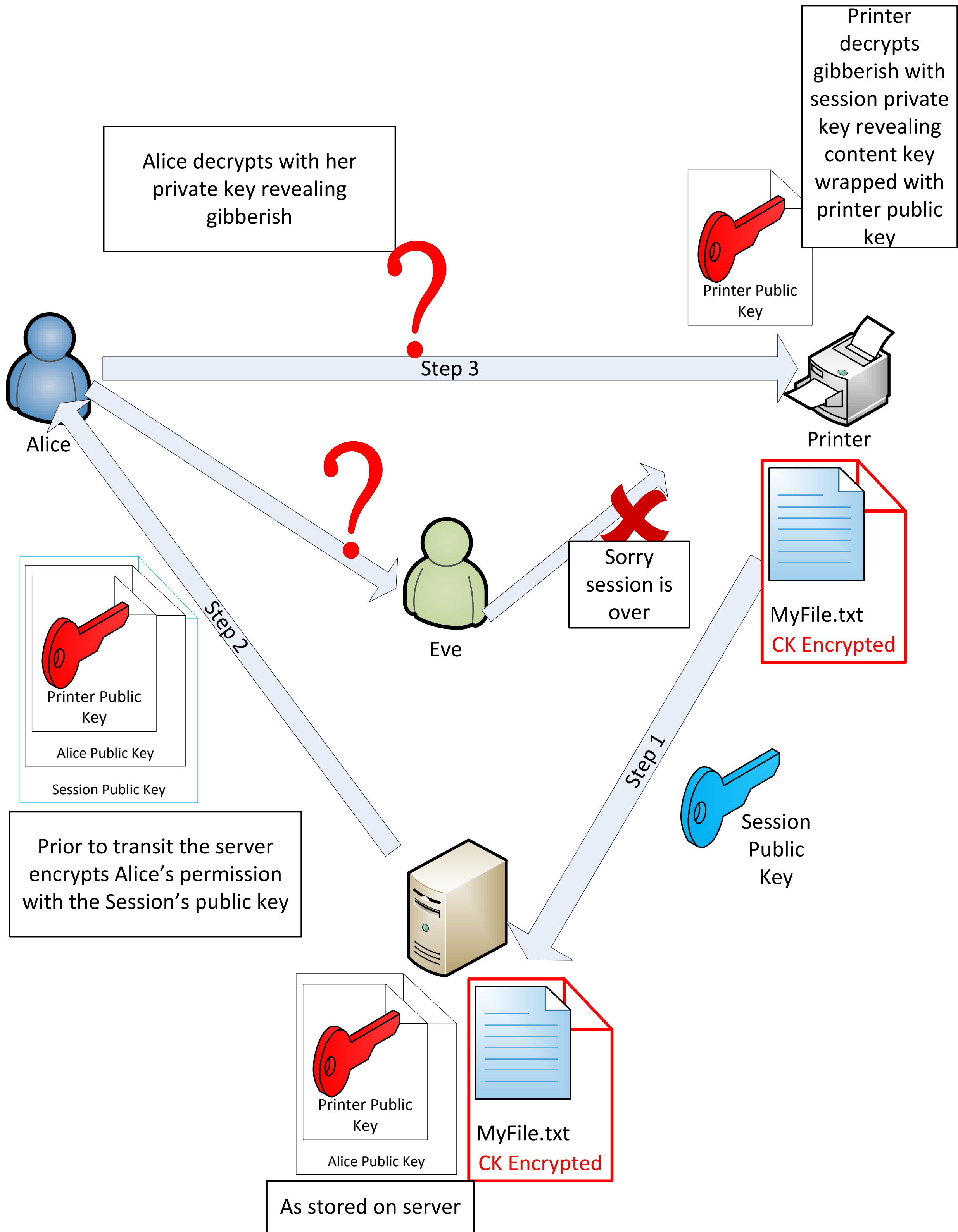

Figure 5

Figure 5 more clearly explains how commutative cryptography is used in SISAP. Let us imagine Alice wants to print a MyFile.txt. In this explanation, the key will be the content key wrapped with the hardware device's public key. This is because at the last step before revealing the content key is the hardware device decrypting it. This never changes so this step can be overlooked. Step 0) Alice communicates her intent to print MyFile.txt with the server and a specific printer. Step 1) The printer will generate a set of public and private keys for the session, and send the session public key to the server. Step 2) The server will use the session public key to encrypt Alice's printer key permission. The blue outlined cypher text $\mathrm{C} = \mathrm{e}_{\mathrm{Session}}\mathrm{e}_{\mathrm{user}}(\mathrm{key})$ will be sent to Alice. Step 3) Alice decrypts the cypher text using her key, revealing $\mathrm{C} = \mathrm{e}_{\mathrm{Session}}(\mathrm{key})$. Then she sends this to the printer. Step 4) The printer decrypts using the session private key which was created earlier. Assuming Alice was able to correctly decrypt her part; the printer will be able access the content key and print the file. This enables SISAP to check the user's identity without revealing any information to the user.

4 Changing Permissions

4.1 Ownership

The creator of the file and any subsequent owners will have access to the content key. This allows owners to grant access to other users. This also allows owners to elevate other users to become owners. Owners have a lot of access; and therefore are considered trusted. While owners have the ability to grant access to any hardware device, they do not inherently have access to every device by default.

4.2 Adding an Owner

When an owner elevates another user to be an owner, the following will happen:

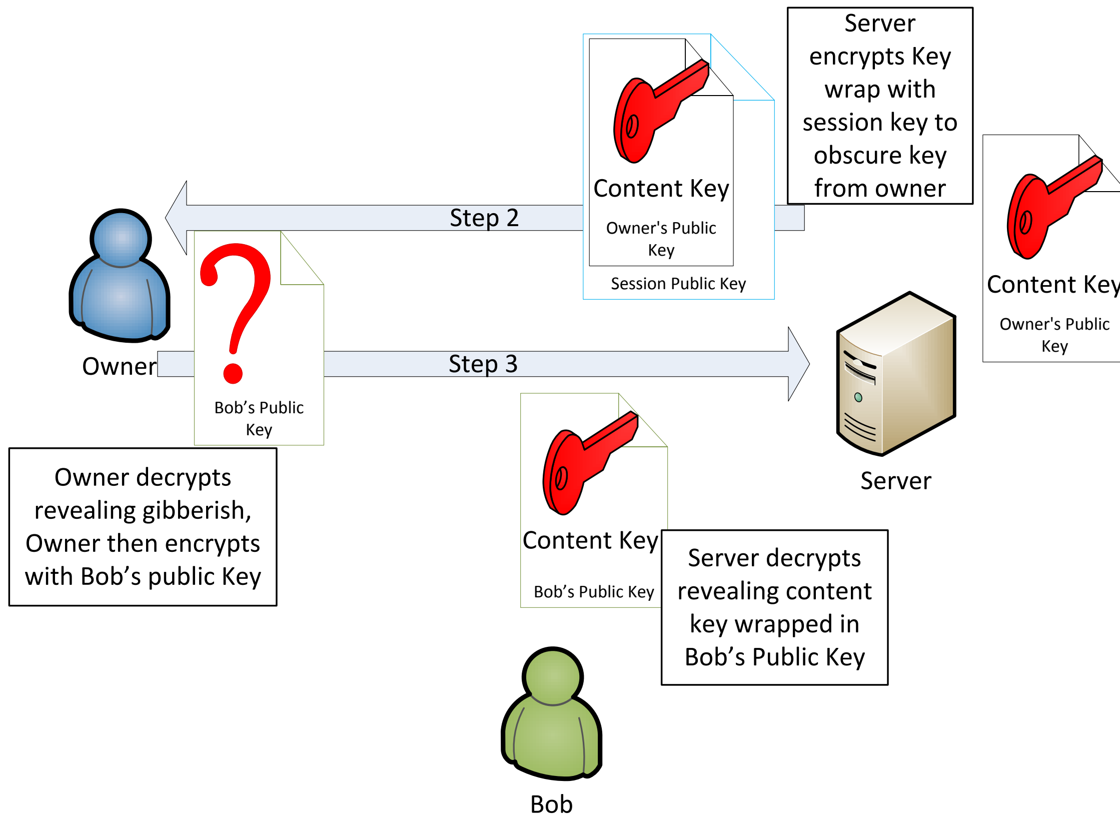

Figure 6

Step 0) The owner will request to elevate a user to become an owner. Step 1) The server finds the owner's permission which is the $\mathrm{C} = \mathrm{e_{owner}(key)}$. The server creates a pair of session keys and encrypts the owner's permission with the session public key, creating $\mathrm{C} = \mathrm{e_{Session}e_{owner}(key)}$. Step 2) The owner will decrypt the cypher text and encrypt it with Bob's public key, revealing $\mathrm{C} = \mathrm{e_{Bob}e_{session}(key)}$. Step 3) The server will decrypt using the session key, revealing $\mathrm{C} = \mathrm{e_{Bob}(key)}$. This allows Bob to become an owner.

By using a session, this allows the owners to manipulate the content key without ever seeing it, therefore, adding a layer of protection to the system.

4.3 Adding a Permission

Adding a standard permission to allow a user to print is very similar to adding an owner. The only difference is you encrypt using the device's public key prior to encrypting with the user public key.

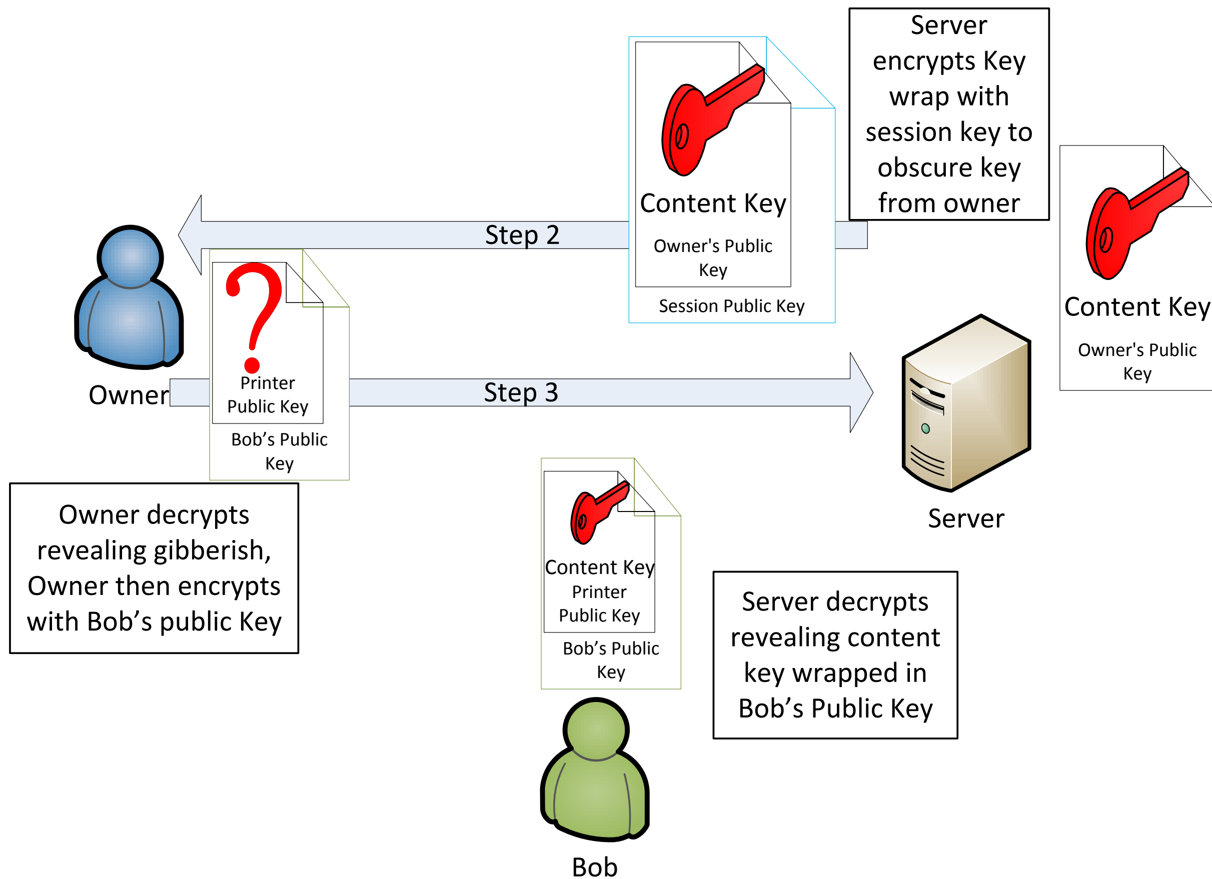

Figure 7

Step 0) The owner will request to add a user permission. Step 1) The server finds the owner's permission which is the $\mathrm{C} = \mathrm{e}_{\mathrm{owner}}(\mathrm{key})$. The server creates a pair of session keys and encrypts the owner's permission with the session public key, creating $\mathrm{C} = \mathrm{e}_{\mathrm{Session}}\mathrm{e}_{\mathrm{owner}}(\mathrm{key})$. Step 2) The owner will decrypt the cypher text and encrypt it with the printer's public key then encrypt it with Bob's public key, revealing $\mathrm{C} = \mathrm{e}_{\mathrm{Bob}}\mathrm{e}_{\mathrm{Printer}}\mathrm{e}_{\mathrm{session}}(\mathrm{key})$. Step 3) The server will decrypt using the session key revealing: $\mathrm{C} = \mathrm{e}_{\mathrm{Bob}}\mathrm{e}_{\mathrm{Printer}}(\mathrm{key})$.

4.4 Malicious Owner

Part of our threat model assumes that the owner is not malicious. This is a fair assumption since the owner will be the one that adds the file to SISAP. However, with the above process, an owner could trick the server into revealing the content key. Imagine the owner receives $\mathrm{C} = \mathrm{e}_{\mathrm{Session}}\mathrm{e}_{\mathrm{owner}}(\mathrm{key})$, and then partially decrypts the key revealing $\mathrm{C} = \mathrm{e}_{\mathrm{Session}}(\mathrm{key})$. Instead of encrypting that with another user's key, imagine if the owner sent that directly back to the server. The server would unwittingly decrypt content key in plain text. Obviously, to mitigate that situation, the content keys could be created to follow a format, and if the server decrypts something that matches, then it would disregard it and flag the owner as malicious.

4.5 Deleting Permissions/Revocation

Unfortunately, revocation, unlike granting access, cannot happen fully encrypted. Therefore the first step is to verify the owner's identity. Every file will have a unique owner ID. Owners prove their ownership by revealing the content of that unique ID. Once an owner is verified, he will be able to perform revocation.

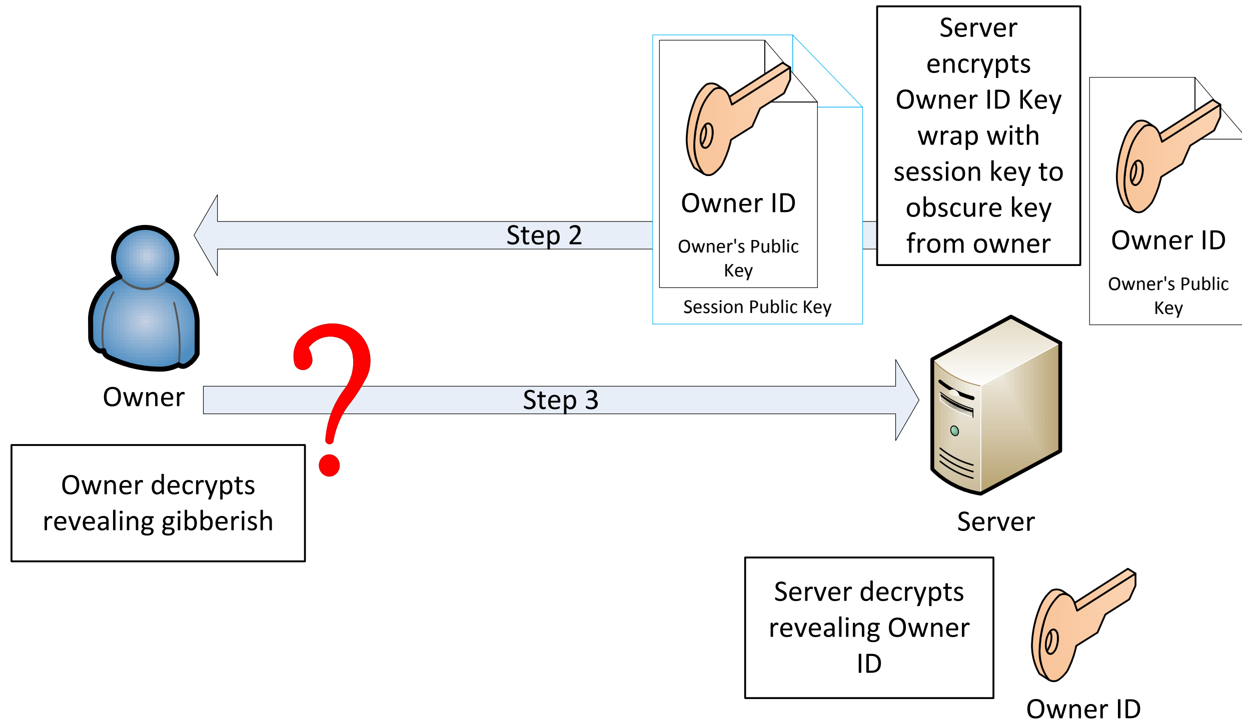

Figure 8

The diagram shown above explains the process of verifying an owner's identity with the use of an owner ID. Step 0) Owner requests to be verified. Step 1) Server finds the owner's owner ID permission and encrypts it with the session public key creating $\mathrm{C} = \mathrm{e}_{\mathrm{session}}\mathrm{e}_{\mathrm{owner}}(\mathrm{Owner\ ID})$. Step 2) The owner will decrypt the received cypher text revealing $\mathrm{C} = \mathrm{e}_{\mathrm{session}}(\mathrm{Owner\ ID})$. Step 3) The server will decrypt using the session's private key, revealing the owner ID. If the revealed owner ID matches the server's version, the user will become verified. As stated above, revocation cannot be done in an encrypted fashion, however a verified owner cannot abuse that status by granting access.

The next step is to process the revocation commands. There is no easy solution for this. There are two approaches.

The first is the most direct method of deleting the key associated to that user's permission. This is secure against malicious users for two reasons. Firstly, the user's permission key must be used in conjunction with a session key, so even if the user has a copy of the key the user by

itself cannot decrypt anything. Secondly, the user should never be exposed to her key, as she will only ever be exposed to the key still encrypted by the session key. Therefore, assuming the server actually deletes the right key, this would be secure. Unfortunately, now we need to figure out how to actually ensure the server deletes these permissions.

The second method is the more traditional approach of re-encrypting the file with a new content key, and then redistributing access to the remaining users. The problem with this is that the system relies on the use of encryption to enforce policy. So that means there is really know way of "knowing" who has access to what. Granted, each key will have a plaintext header that describes the key. But there is really no way of knowing that the key actually belongs to who the header says it belongs to. So the solution is to find how to grant access to the remaining users after a new content key is created. For this to work, we need to ensure that the plaintext permission headers set has not been tampered with.

Of these two situations both have potential downfalls when combined with a malicious server. In the first case, the server may not delete the user's permission, potentially extending a user's access. In the second situation the server could add users to the plaintext header, and potentially allow any user access to the file. Obviously, neither of these situations is acceptable in their current form of implementation.

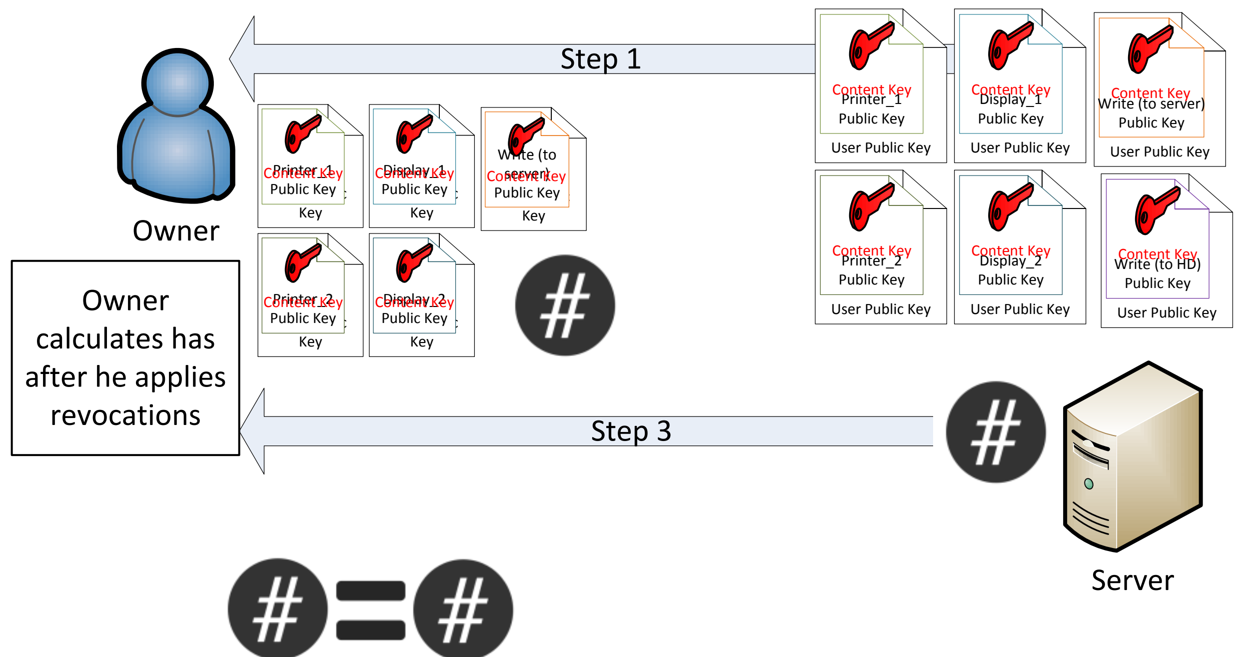

Figure 9

In SISAP, we chose to modify the first method of purely removing the permission associated with the user since a malicious server can do less damage. To verify the server deleted the permission a hashing function is used. After the owner has been verified, the owner can begin

the revocation process. Step 0) The owner requests a certain user's permission to be revoked. Step 1) The server sends entire set of permissions related to that file to the owner. Step 2) Owner applies revocation and calculates a hash on remaining permission set. Step 3) Server applies revocation, calculates hash and sends hash to owner. Step 4) Owner determines if hashes match, and if they do match, the owner can know with confidence that the server applied the revocation.

5 Future Directions

5.1 Integration with an array of devices and applications

In this paper, the main hardware devices described was a printer. To extend this to be used in a full modern office environment, more hardware devices will need to be incorporated. From an understanding standpoint, it would not be hard to just replace the printer with a monitor, for example. However, these devices (including printers) do not just accept files. Most printers accept a raw bitmap and most displays accept a stream of colors. Obviously, physically implementing SISAP would require changes to the drivers we have today and perhaps even changes to the programs we use. However, changing many programs that have been written does not seem like a feasible solution. Since processing the file can only be done in plain text, the file must be decrypted before being processed. Therefore there needs to be some bi-directional communication between hardware devices and the programs that are accessing and editing the files.

5.2 GPS and time

Another extension to SISAP could be adding additional security features like GPS and time parameters, allowing further fine-tuning of access control. These parameters would allow a location and time-based policy to be implemented. However, implementing this with encryption, while remaining secure, is still being developed. Potentially, this could be done by embedding information into the session keys or creating more layers for the user to decrypt. This would solve the problem of time revocation as the session would periodically expire.

6 Conclusion

SISAP has been able to achieve many desired features. It was able to create an encryption scheme that does not rely on a server making a simple Boolean decision regarding policy. This allows the server to be less trusted. Furthermore this allows policy to be enforced with encryption which means it is must harder to fake. Neither the user nor the server is considered a trusted device in this system but rather the end-point hardware devices are trusted. This means that even if a single device is compromised only data allowed at that device can be

intercepted. More understanding of encryption based policy has come to light with the research of SISAP.

However, there is still much to be done. The actual implementation with programs and hardware devices is only possible in theory. Hardware devices and drivers must be reengineered in order to allow for SISAP to become a reality. Unfortunately, much more research is needed before even prototyping can begin.

7 Acknowledgements

This research could not have been done without the knowledge and insight of my mentor Mike Vai. Also Nabil Schear provided valuable insight regarding encryption and key management. I would also like to thank MIT Lincoln Laboratory for providing the facilities and money that allowed me to complete this research.