

CS6903 - NYU - Fall 2015

Project 2 - Andre Protas (ADP369) & Nate Rogers (NJR5)

GISTER: Secure Dead Drop Using GitHub

Abusing Sharing Service for Encrypted Comms



TABLE OF CONTENTS

[Introduction](#)

[Components](#)

[GitHub Gist](#)

[Python](#)

[Implementation & Threat Modeling](#)

[Message Security & Integrity](#)

[AES Key](#)

[Message Structure](#)

[Decryption Process](#)

[Message Security Threat Model](#)

[Application Security](#)

[Unpredictable IDs & Metadata](#)

[Revision history](#)

[Red Herring Support](#)

[Publicly Viewable](#)

[Disadvantages](#)

[Application Threat Model](#)

[Network Security](#)

[Network Security Threat Model](#)

[Usage](#)

[GISTER_TRANSMIT.py](#)

[GISTER_RECEIVE.py](#)

[Potential Improvements](#)

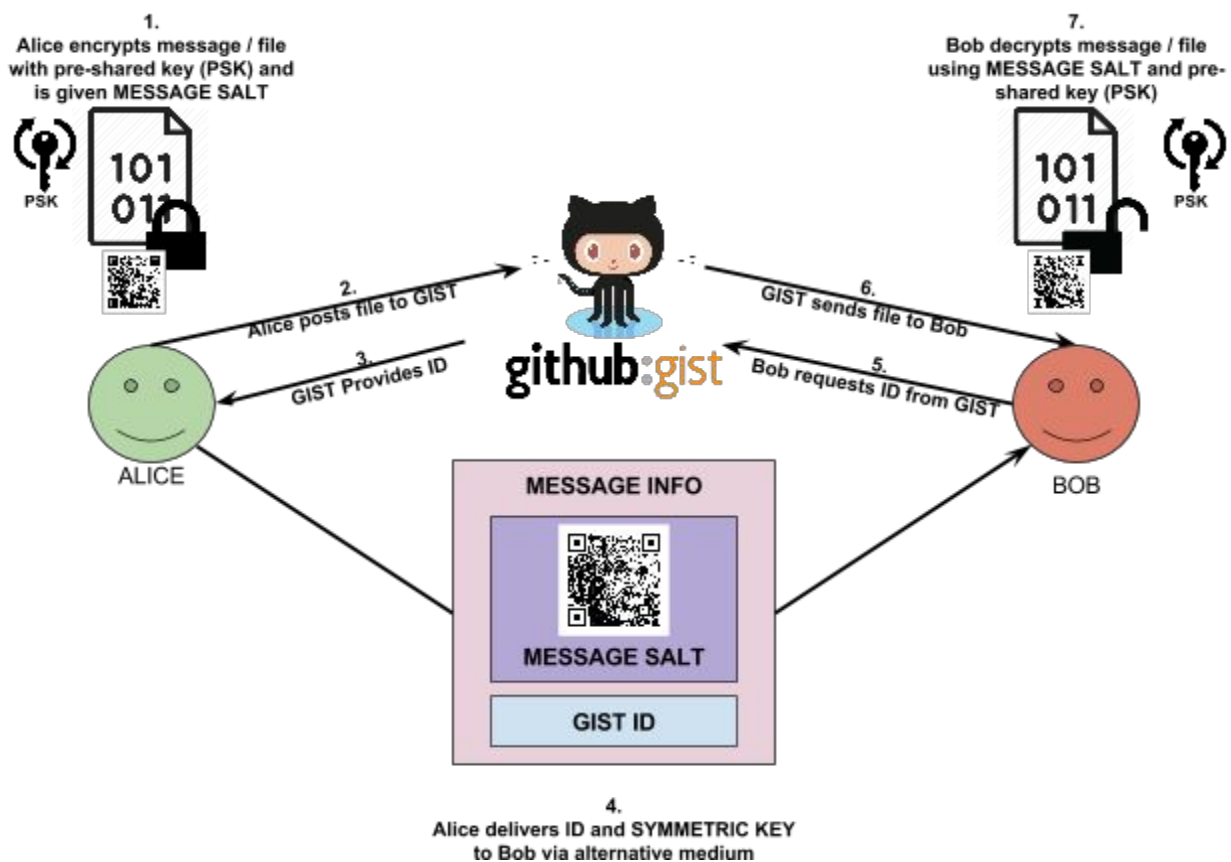
[Summary](#)

Introduction

Our project utilizes a publicly-available and anonymous dead drop service as a means of hosting strongly encrypted messages. This allows a user to transmit a message (or file) to another user anonymously, and then share the file location and necessary cryptography material using an alternative medium (in our case, QR code) to enable the user user to identify, download, and decrypt the message.

While this is not an entirely new concept, we believe that our implementation has several concepts that are able to leverage some of the concepts from class, general cryptography best-practices, as well as general operational security (OPSEC) principles.

A high-level overview of our implementation is depicted below:



Components

Our project focused on the concept of an anonymous internet dead drop website. This site allows any unauthenticated user to create a page and post data to it. If another user knows that the page exists, they are able to see it without needing to authenticate themselves (i.e., both parties can use the site anonymously). While this gives us many advantages from a privacy perspective, it is a challenge from a cryptographic perspective in that you have no way to authenticate the user with the site functionality, and you must rely on the message authentication.

GitHub Gist

For our project's dead-drop service, we utilize GIST.GITHUB.com. This site offers us the following benefits:

- Well-documented REST API¹ (simple to prototype and interact with)
- Does not require account creation (supports anonymity)
- Uses SSL for network security (we verify SSL chain)
- Allows for large file sizes (up to 10mb)
- Files can be easily retrieved by any user (dead drop)
- Files cannot be deleted by others² (preserves integrity)
- Posting base64 files to GIST service is normal practice (hide in the noise)
- Reputable API.GITHUB.com domain (not abnormal network traffic)
- Revision history shows if files are manipulated (preserves integrity)

The only network traffic our code generates is to API.GITHUB.com, which provides all of the Gists functionality. All traffic is over SSL, so any passive observer will simply see data being transferred between the client host and API.GITHUB.com, without knowing that it is accessing the Gist functionality.

¹ <https://developer.github.com/v3/gists/>

² <https://help.github.com/articles/deleting-an-anonymous-gist/>

Python

Our project is implemented entirely in python. This allows it to be easily used in multiple environments (we tested against Ubuntu and Windows 7 without issues).

To support all of our components, we used reputable public libraries. The following 3rd party libraries were the only non-native code that we used:

- pyaes³: A well-known crypto library to provide encryption/decryption services
- requests⁴: An HTTP client that supports advanced SSL validation and verification
- pyqrcode⁵: A simple QR code generator library

Python pip is able to install all of the libraries with minimal effort (i.e., `pip install <package name>`).

All other functionality is native to Python and requires no further installation. We tested against python 2.7.10, and assume backwards compatibility to python 2.7.8 or greater.

³ <https://github.com/ricmoo/pyaes>

⁴ <http://docs.python-requests.org/en/latest/>

⁵ <https://pypi.python.org/pypi/PyQRCode>

Implementation & Threat Modeling

Our project focuses on adding cryptographic security at three separate layers: the Network, the Application (Gist), and at the generate message packages. Each of our security-focused implementation details are described in the following sections.

Assume Open Source: We do not use any “security by obscurity” concepts, and all decisions are made to challenge an attacker’s ability to decrypt a message assuming they have the transmitted message and source code.

Secure Deletion: Some data is written to disk, such as the QR codes for displaying to the user in a simple way (via a web browser). After being displayed to the user, they are securely deleted by (1) overwriting the data with random bytes, (2) setting the size of the file to 0, and then (3) deleted. This helps us avoid file-based forensics to identify transmitted message records.

Out-Of-Band Comms: Since we are utilizing a publicly accessible dead drop, it is critical that our two parties have some additional method out-of-band method of communication. There are three pieces of data that must be shared to be able to decrypt a message:

- Pre-Shared Password: can be shared in person for initial message, and changed over a follow-on secure message
- Message-Specific ID: a Gist ID that is used to retrieve the message from GitHub Gist servers
- Message-Specific Key Salt: the salt that is used with the pre-shared password to generate the message-specific AES key

Thoroughly Tested: Our crypto scheme comes with a test suite that exercises all critical message, application, and network code. We manually tested the suite on multiple hosts, and feel confident in it’s reliability. Our test suite takes ~45m to run a full battery.

We describe these in a network-based top-down approach, starting with message security (also the most difficult) through the network layer (the lowest and easiest from our perspective).

Message Security & Integrity

Our messages are secured using AES-256 encryption. AES-256 is considered a strong encryption scheme and is sufficient for our purposes. To avoid a user choosing an insecure password, we combine a randomized salt and simple transmission method (QR code) to make it easier for users to share high-entropy and long key material to each other.

AES Key

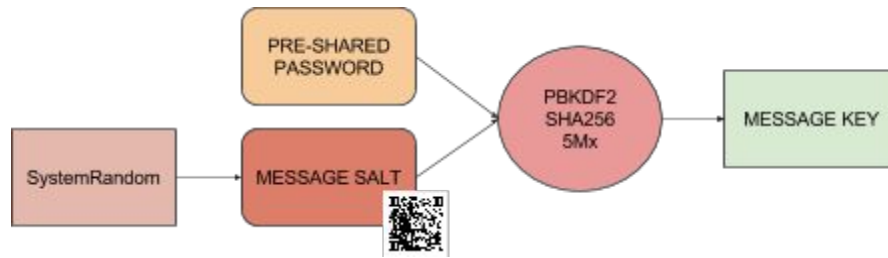
The AES key is generated using the pbkdf2 (password-based key derivation function; PKCS #5) method included within the python hmac library⁶. This function requires the use of a password and a salt, and the output is the key that is used to encrypt the message. We perform 5 million iterations of the PBKDF2 algorithm to make it more difficult to derive a key if an attacker is able to intercept either the pre-shared password or the message salt. The additional iterations also provide the benefit of preventing timing attacks where an adversary could guess the key based on the time required for a valid or invalid response.

The password is pre-shared between the two parties in advance (using an out-of-band channel such as a phone call or text message). The salt is generated for each transmission, and utilizes the most secure OS random functionality⁷ available to python depending on the OS that it is running on. This salt is then transmitted to the receiving party out-of-band, similarly to the pre-shared passcode. To assist in the transmission of this IV, a QR code is generated and displayed to the user.

This process is depicted below:

⁶ https://docs.python.org/2/library/hashlib.html#hashlib.pbkdf2_hmac

⁷ <https://docs.python.org/2/library/random.html#random.SystemRandom>



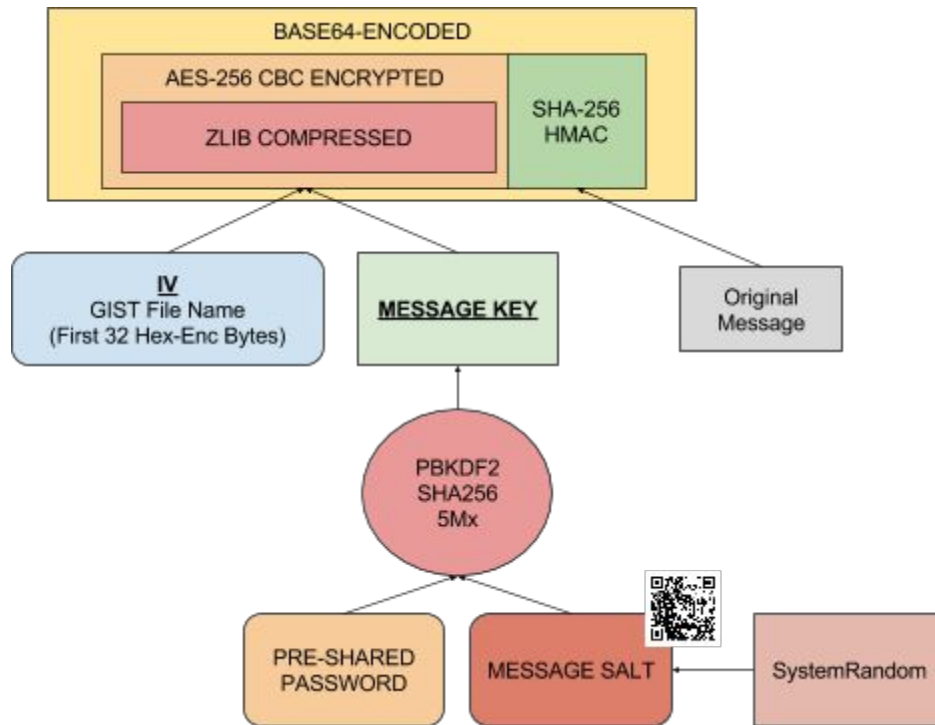
Message Key Generation Components

Message Structure

Prior to acting on the message, we take a SHA-256 HMAC. This is saved in memory, and provided to the recipient in the clear to validate the integrity of the message once decryption and decompression has taken place.

We utilize AES in cipher block chaining (CBC) mode to challenge cryptanalysis. The CBC initialization vector (IV) is provided in the clear as part of the message metadata (using the first 32 hex characters of the file name as a feature offered by Gist API). This IV is combined with the Message Key (described earlier) to encrypt the message. The message is first compressed (compression is near useless on a high-entropy dataset, such as an encrypted message) and then encrypted using the Key and IV.

The components and process is depicted below:



Message Generation Components

Decryption Process

Upon receiving the message, the following steps are taken by the decryption script:

1. Base64 decode the candidate message
2. Strip off 16-byte HMAC and save for validation
3. Store 32-byte IV from metadata (file name)
4. Attempt decryption on remaining message (HMAC removed)
5. If no decryption errors detected, attempt decompression on potentially-decrypted message
6. If no decompression errors, validate HMAC of decompressed message against provided HMAC
 - a. Use secure `compare_digest` method provided by HMAC library⁸ to avoid timing attacks

If the message HMAC is found to be identical, then we have successfully decrypted the message and it is written to disk.

Message Security Threat Model

- Scenario 1: Actor Has Message & Code
 - Since we are using a publicly-accessible dead drop side, our threat model assumes that an adversary has access to both the code used, and the messages. For message security, we want to make decryption as challenging and slow as possible for a brute-force attack. Decryption would require brute forcing across the entire key space for a compressed, unknown message.
- Scenario 2: Actor Has Message, Code, and Salt
 - Based on our 2-part message key generation and substantial processing time (5 million repetitions), breaking the message even if the the message salt was known would be unrealistic. Even if an attacker could choose his own ciphertext and brute force the system any observable time difference

⁸ https://docs.python.org/2/library/hmac.html#hmac.compare_digest

between valid and invalid responses would be extremely difficult to detect thus making this timing based attack with a chosen ciphertext ineffective.

- Additionally, since we are generating a per-message salt, the key used in one message can not be used to decrypt another message unless the new message's salt was also intercepted.

Application Security

Because we have chosen a specific web service to interact with, we feel comfortable utilizing some of the functionality of this service to help secure our communication channel. The characteristics of the site offered us several advantages.

Unpredictable IDs & Metadata

We cannot choose an ID even if we wanted to, as they are non-sequential and provided by the API. For an attacker to attempt to identify messages that belong to our encryption scheme, or more specifically, messages between two parties using our encryption scheme would be incredibly difficult and would more likely require GitHub collaboration and temporal analysis.

There are multiple types of metadata that are provided by the API, such as file names, descriptions, and public accessibility. To avoid a party from fingerprinting our messages and searching the Gist database for all possible transmissions, we randomize all metadata content. This makes it very difficult for an attacker to identify messages by searching through the metadata. The metadata options are also used for transmitting the IV.

Revision history

A message can be altered by any other anonymous party, but all changes are recorded as revisions. There is no reason another user would legitimately modify our messages, so if more than one revision is detected, the recipient is notified of the potential tampering and no longer processes the message. We considered utilizing this method to perform an acknowledgement of the message, but we decided that it would reduce the security of the message from it's original state, and it would be just as easy for the recipient to

generate a new message and send it to the original sender acknowledging the previous message within its contents.

Red Herring Support

We are only transmitting one message per ID. However, Gist allows for multiple files. Once we create our encrypted message, we then create a random number of other messages of the same length and entropy (random) and post them as part of the same ID. This makes it more challenging to search for Gist ID's used and our schema, and more difficult for an attacker if they identified our message, as they would need to identify which file was the actual transmission. For a recipient that is able to generate the proper message key with the salt and pre-shared password, this requires a little extra processing but is negligible. There is no way to discern which message is the real, decryptable message (even to the intended recipient). Lastly, to prevent timing based attacks against our red herring concept, we ensure that we attempt to decrypt every message regardless if a correct response has already been found. This way an adversary cannot attempt to brute force the system by measuring the difference in processing time between valid and invalid message decryption.

A depiction of this is below:

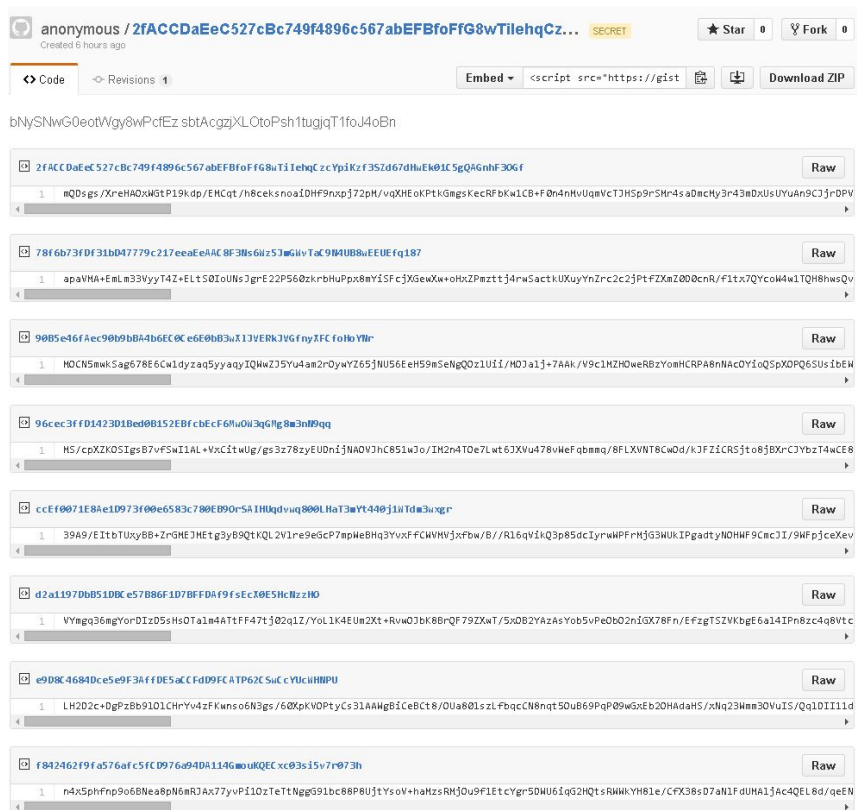


Red Herring Depiction

Publicly Viewable

One characteristic that doesn't necessarily offer any advantage to our cryptographic scheme, but did make it easier to debug our code was the ability to visually review a Gist posting without requiring code. This is the primary intent of the site (to share code), and was used by us extensively.

Below is a picture of one of our messages as posted to Gist:



Publicly-Accessible View of Message

Disadvantages

The biggest disadvantage of the system is the inability to delete a message at a later time. However, this isn't a substantial issue to us, as there are so many messages on Gist. This also would complicate a one-to-many scenario whereby you'd like to send one message to multiple recipients that all have the same pre-shared password.

Application Threat Model

The following scenarios were focused on as part of our encryption scheme as it pertains to the application usage.

- Scenario 1: Actor Searches for All Messages
 - We avoid having our messages be identified en masse by randomizing all metadata. Additional, our usage of Base64 data makes it difficult to overtly identify only our messages. Large-scale searching would be required to find highly entropic messages underneath base64 encoding, which would require massive search and processing capabilities and likely collaboration with GitHub to collect enough data.
- Scenario 2: Specific GitHub Message ID is Identified
 - If a specific ID is identified (perhaps as captured via out-of-band transmission), the Message Security layer and additional red herring messages will stand up to cryptanalysis (nothing is given away). We assume that an actor can identify our messages.
- Scenario 3: Actor Modifies Message
 - If an actor modifies a message (supported by GitHub API), then a revision history item will be appended. The only possible way to avoid this would be with collaboration with GitHub personnel, which would also require the knowledge of a message to target prior to it being received by the intended recipient.
- Scenario 4: Actor Replays Message
 - An actor is free to replay a message as it stands (copy all files and create a new post). This is the nature of a dead drop since it allows for anonymous dropping of data. However, the out-of-band nature for notifying a recipient of a message being available would not be known to an outside attacker.
 - In order for an attacker to be able to replay a message, they would need to know the ID of a message, the out-of-band transmitted salt, and be able to masquerade as the sender (e.g., same phone number for SMS, etc) to be able to only send the same message. As long as a reasonably secure out-of-band communication channel is selected, this is untenable.

Network Security

Network security is perhaps the easiest layer of security within our project, and relies simply on SSL. Since GitHub uses SSL, much of the work is already done for us. Our implementation effectively performs the following:

- Only connects to API.GITHUB.com via an SSL connection (by using HTTPS)
- Do not allow redirects to a potentially unencrypted site
- Validate that SSL is being used
- Validate that the certificate for API.GITHUB.com has a valid chain to known CA roots
- Carry our own CA root for DigiCert (the certificate used by GitHub) bundle to avoid an advanced adversary that may have a potentially rogue CA root
- “Fail-closed” in case any SSL errors are detected to avoid transmitting any potentially-sensitive traffic on a potentially insecure network link
- Allow for proxies to increase privacy (privacy is not really in scope for this project, so this was not a focus)

Our implementation focuses on the python requests library. This library has all of the necessary capabilities to create SSL connections to web servers, and can enforce SSL certificate chains to know certificate authorities through the use of a CA_BUNDLE file. We created a CA_BUNDLE file by using the CURL CA_BUNDLE⁹ and removing all non-DigiCert CA certificates.

As requests makes connections to API.GITHUB.com, it verifies that the certificate received (1) has a valid certificate chain, and (2) that the root CA certificate matches one of the certificates on within the provided CA_BUNDLE.

The python requests library additionally supports the ability to use a network proxy. This could easily allow a user to tunnel all traffic through TOR or a free proxy to add additional layers of network obfuscation to mask their original IP or avoid PASSIVE monitoring. This

⁹ <http://curl.haxx.se/docs/caextract.html>

is enabled simply by creating an HTTPS_PROXY environment variable¹⁰ within the environment of the user or process.

Network Security Threat Model

Based on the mechanisms that we put in place, the only scenarios that seems possible for eavesdropping on the traffic is the following:

- Scenario 1
 - Malicious actor has ACTIVE MITM access (either via network or DNS redirection) to our network link between our client and API.GITHUB.com
 - Actor is able to sign SSL certificates with a DigiCert root CA certificate chain
- Scenario 2
 - Malicious actor has PASSIVE MITM access and is able to monitor all traffic to API.GITHUB.com
 - Actor is able to decrypt SSL traffic with knowledge of API.GITHUB.com private certificate

While these scenarios “may” be possible, they would require extensive capabilities and access to accomplish. We feel that this raises the bar significantly on the potential for an outside actor to be able to monitor our traffic.

¹⁰ <http://docs.python-requests.org/en/latest/user/advanced/#proxies>

Usage

GISTER_TRANSMIT.py

The transmission usage is fairly simple. All a user needs to do is run the script with one argument: the file to post. Once this is executed, the user will be asked for the a pre-shared password. After performing the encryption and posting, the default web-browser will display the two pieces of information (the GIST ID and the MESSAGE SALT) that need to be transmitted out-of-band to the recipient.

A depiction of this is below:



```
MESSAGE ID:
9c3fa30ef28bcf7dfa27

MESSAGE SALT:
XprwIw45X6GtyutJPBkV1IIRarLVhAcEj4cuU2z179o=

[DEBUG] 2015-12-05 22:42:15.431: Generating HMAC
[DEBUG] 2015-12-05 22:42:15.431: HMAC for data: 4722ce8d07a3e1b3032d32c4bbbed77a
[DEBUG] 2015-12-05 22:42:15.431: Compressing Data
[INFO] 2015-12-05 22:42:15.433: Compressed data from 7824 -> 2606 bytes
[DEBUG] 2015-12-05 22:42:15.433: Encrypting Data
[DEBUG] 2015-12-05 22:42:15.470: Encrypted Data: 2608 bytes (input: 2606)
uploaded 3500 2624
[INFO] 2015-12-05 22:42:15.470: Final file size for upload: 3500
[DEBUG] 2015-12-05 22:42:15.505: Generating File Name
[INFO] 2015-12-05 22:42:15.506: GIST File Name: B759bC18fECdfcAe278C6cBF34dFCF4ewfItkUv4n1muxJtP03d
gZPhkm38SecfCeaQDBdn
[DEBUG] 2015-12-05 22:42:15.539: Generating File Name
[INFO] 2015-12-05 22:42:15.540: GIST File Name: 767EDCf1bB0af4BfC357A3F1Cd4ae9c0g8f93nmYfrulqEG2gyH
Vik90zzPQenn
[DEBUG] 2015-12-05 22:42:15.573: Generating File Name
[INFO] 2015-12-05 22:42:15.575: GIST File Name: 300cACB0A4feFF3Da4A93D7Aaah6DCBG10c1HKTlo0gvi0wxMx
9qDXoUMLTfpj1JHn10Doj1J9AZ2tnl
[INFO] 2015-12-05 22:42:15.584: Starting new HTTPS connection (1): api.github.com
[DEBUG] 2015-12-05 22:42:16.154: "POST /gists HTTP/1.1" 201 17698
[INFO] 2015-12-05 22:42:16.170: GIST ID: 9c3fa30ef28bcf7dfa27
[INFO] 2015-12-05 22:42:16.229: QR code created
[INFO] 2015-12-05 22:42:16.313: QR code created

C:\Users\User\Documents\GitHub\ngyu-poly\CS6903\Project2>c:\Python27\python.exe gister_transmit.py gi
ster_receive.py
```

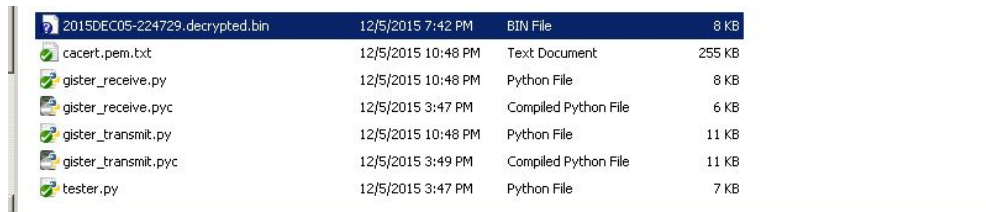
Transmission Example

GISTER_RECEIVE.py

The receiver usage is also very simple. All that needs to be entered as parameters are the GIST ID and MESSAGE SALT as provided out-of-band by the sender.

Once these are entered as parameters to the script, the pre-shared passcode is requested. All actions against Gist are then performed automatically and, provided a message is successfully decrypted, it will be output to the current working directory with a generic name of the date and time.

A depiction of this process is below:



File Name	Date Modified	File Type	Size
2015DEC05-224729.decrypted.bin	12/5/2015 7:42 PM	BIN File	8 KB
cacert.pem.txt	12/5/2015 10:48 PM	Text Document	255 KB
gister_receive.py	12/5/2015 10:48 PM	Python File	8 KB
gister_receive.pyc	12/5/2015 3:47 PM	Compiled Python File	6 KB
gister_transmit.py	12/5/2015 10:48 PM	Python File	11 KB
gister_transmit.pyc	12/5/2015 3:49 PM	Compiled Python File	11 KB
tester.py	12/5/2015 3:47 PM	Python File	7 KB

24729.decrypted.bin Date modified: 12/5/2015 7:42 PM Date created: 12/5/2015 7:42 PM
Size: 7.63 KB

```
passer 2015-12-05 22:47:29.030: Starting new HTTPS connection (1): api.github.com
Please Enter Pre-Shared Key:
https://api.github.com/gists/9c3fa30ef28bcf7dfa27
[INFO] 2015-12-05 22:47:29.030: Starting new HTTPS connection (1): api.github.com
[DEBUG] 2015-12-05 22:47:29.467: "GET /gists/9c3fa30ef28bcf7dfa27 HTTP/1.1" 200 None
[INFO] 2015-12-05 22:47:29.469: Decrypting Message
[DEBUG] 2015-12-05 22:47:29.470: Candidates for Decryption: 4
[DEBUG] 2015-12-05 22:47:29.470: Attempting Decryption on Candidate 1 of 4
[DEBUG] 2015-12-05 22:47:29.470: Message Length: 1 (bin: 2624)
[ERROR] 2015-12-05 22:47:29.503: Unable to Decrypt
[DEBUG] 2015-12-05 22:47:29.503: Did Not Decrypt
[DEBUG] 2015-12-05 22:47:29.503: Attempting Decryption on Candidate 2 of 4
[DEBUG] 2015-12-05 22:47:29.503: Message Length: 1 (bin: 2624)
[ERROR] 2015-12-05 22:47:29.538: Unable to Decrypt
[DEBUG] 2015-12-05 22:47:29.539: Did Not Decrypt
[DEBUG] 2015-12-05 22:47:29.539: Attempting Decryption on Candidate 3 of 4
[DEBUG] 2015-12-05 22:47:29.539: Message Length: 1 (bin: 2624)
[ERROR] 2015-12-05 22:47:29.572: Unable to Decrypt
[DEBUG] 2015-12-05 22:47:29.572: Did Not Decrypt
[DEBUG] 2015-12-05 22:47:29.572: Attempting Decryption on Candidate 4 of 4
[DEBUG] 2015-12-05 22:47:29.572: Message Length: 1 (bin: 2624)
[DEBUG] 2015-12-05 22:47:29.608: Decrypted data (2606 bytes)
[DEBUG] 2015-12-05 22:47:29.608: Decompressed data (7824 bytes)
[INFO] 2015-12-05 22:47:29.608: Successfully decrypted message. Len: 7824
[INFO] 2015-12-05 22:47:29.608: Final Package Decrypted: 2015DEC05-224729.decrypted.bin
G:\Users\user\Documents\GitHub\nguy-polu\CS6903\Project2>
```

Receiving Example

Potential Improvements

Throughout the project we came up with new ideas and implemented them. A few concepts were considered, but not implemented due to time and complexity restraints. Additionally, although they might offer some slight improvement, we did not think that they were substantial. Regardless, we mention them here as consideration if this project was to ever be improved upon, by ourselves or others.

- Using a GIST ID for a long-running conversation
 - The GIST ID could be modified over time by both parties to avoid needing out-of-band contact for each transmission. Almost like a “chat” ID.
 - For our scheme, and since GIST ID's are “cheap” (free to register for new ID's), we went with the model of building one ID for each message and avoiding having one ID show too much information.
- Transmit the next MESSAGE SALT in each transmission
 - This would avoid the need for the SALT to be transmitted out-of-band for each message. However, it then puts a dependency on each message chain being grabbed (in order to decrypt the next message).
 - For our purposes, we treated each message as an atomic action.
- Protect receiving IP address with multiple requestors
 - If the recipient is in a potential area where their IP revelation would cause concern (requiring GitHub collaboration to identify the specific ID being requested for a network attempt), automation could take place to request the Gist ID from multiple hosts over time. This would effectively “hide” the real recipient once they made the legitimate request, potentially from an attributing IP address. These hosts could be on VPS, or TOR clients, anything that would be able to cause enough traffic to the Gist ID that a legitimate request would not be easily identifiable.
 - For the intent of this project, this wasn't the primary focus so we didn't exploit this potential improvement.
- Expiration for messages
 - Because all messages on GIST are just Git repos it is possible to remove specific commits via the ***git rebase***. For our implementation we decided

this wasn't required due to the high volume of messages on the site. With enough messages it would still be relatively easy for two parties to communicate undetected in the normal traffic ("Hide in the noise"). Using this functionality also requires an authenticated user, so it removes the anonymization advantage.

- For our project, we didn't want to add complication and also force users to create legitimate accounts to be able to use our software.
- Increased number of red herring messages
 - We debated increasing the number of fake, "red herring" messages generated for each message but ultimately decided against. We felt the rest of our implementation is cryptographically sound and that more than 8 fake messages may needlessly slow down the system and potentially raise flags for the Gist DevOps team. We were unable to come up with a method for proving this is the case, or justifying increasing the count, but it's likely that a reasonable increase would not cause any issues or unjustified slow-down..

Summary

We believe that GISTER is an example of a useful and properly engineered method for communicating securely over a website dead drop. Multiple layers of protection are implemented to challenge even an advanced cryptanalytic department, and would require extreme processing power and likely collaboration at multiple telemetry points to be able to effectively decrypt any communication.

We believe that the if GISTER was used in a real-world setting, any adversary attempting to attack it's scheme would need to consider alternative methods of access, such as trojanizing the sending or receiving hosts. We feel that attacking the GISTER scheme itself is not a realistic cost-benefit threat model outcome, and thus would be a valid utility for secure and semi-covert communications.