Guaranteed Ransomware

William Sun, Elaine Wong and Brian Schuster

December 22, 2017

1 Introduction

Ransomware is a form of virus where an attacker encrypts a victim's files, and extorts the victim to get the decryption key back. However, there is no guarantee that the victim will get back their key when paying. We present an implementation of "guaranteed ransomware," which guarantees to the victim that the ransomer will only get paid if the ransomer provides them with the correct key. To do this, we used Ethereum, a blockchain hosting access to a decentralized Turing-complete virtual machine, to impartially solve this **guaranteed ransomware problem**.

Project prompt: Can ransomware be made better? Current ransomware payments don't always guarantee that you'll get the key to decrypt your files. Can you design a conditional payment mechanism for Bitcoin or Ethereum that guarantees you will get the key – or the payment does not go through?

2 Disclaimers

The code written here is created strictly for academic purposes, as a proof of concept. Furthermore, it contains many limitations and flaws (both by design and for ease of implementation) that prevent it from being effective for real world usage. The code and ideas here should not be used in any way to implement real ransomware.

3 Assumptions

1. **The victim has no knowledge that they will be attacked, and cannot take any action to prevent or interfere with an attack.** Theoretically, malware protection could protect against potential ransomware attacks, but anti-malware-protection could counter it, but we will not be concerned with virus protection for this paper.

2. **The memory, instructions, etc. that were executed during the ransomware attack cannot be recovered or retraced by the user.**

Otherwise, the ransomware attack would be pointless, since the user could recover the key instead of paying the ransom.

3. **The ransomer properly encrypts, such that the correct decryption key will result in the victim's original files, and the MACs match up.** Presumably, the ransomer is incentivized to encrypt this way, because the attack will be widespread, and they want public trust of their ransomware.

4. **The communications between the client and server cannot be intercepted or interfered with.** There are likely ways to mitigate this (public key system, non-static URLs), but for ease of implementation we make this assumption.

4 Design

The source files for this project include code for a **server**, and code for a **client**.

1. The **server** is owned by the ransomer, assumed to be continuously running. It connects to the blockchain via the geth (Go Ethereum) tool.

2. The **client** code is only run when the victim gets ransomed and the victim interacts with the server to decrypt their files. It provides instructions for the victim to set up an Ethereum wallet, purchase ether, and pay into the smart contract ransom. Furthermore, it is fully auditible by the victim and provides the victim with a copy of the smart contract's source code.

For ease of setup, the server and client run on the same computer in this implementation. However, it is trivial to run the server remotely from the client.

The algorithm for the encryption and decryption process, that "guarantees" the ransom, is as follows:

1. **Encryption.** The victim's unencrypted file is hashed with SHA-256 as a MAC. Then, with a random key, the file is encrypted with the Caesar cipher.

2. **Decryption.** The encrypted file is decrypted using the key, and the decryption is verified by taking the SHA-256 of it.

5 Flow

The intended flow for project to be used is as follows:

1. The ransomer has their server and a smart contract up and running.

2. The victim triggers the ransomware.

3. The ransomware hashes the contents of a folder in the victim's computer, generates a random key, and encrypts the contents of a folder with that key.

4. The ransomware sends the hash and key to the ransomer's server, and deletes the key. The hash is used by the ransomer as a unique identifier for that victim.

5. The victim is prompted to create an ether wallet, and use it to send their data and some ether to a contract. The user is also invited to audit the code of the smart contract and the code of the client-side decryption process code.

6. The victim's payment of the ransom to the wallet triggers the ransomer to send their key to the smart contract.

7. Having received the key, the smart contract calculates the decryption of the data, and sees if its hash matches the one originally provided by the victim. If the decrypted plaintext is valid, it sends the ether to the ransomer's address, and the key to the victim. If it is not valid, it does nothing.

8. After the ransomer receives their money, they reveal the key to the hash on their server, allowing for the client's program to pull the key and automatically decrypt. While this is not strictly secure, by inspecting the blockchain and the contract's event logs, the victim can get the real key; this is just for the convenience of the victim.

A video of the program in action can be found at `https://docs.google.com/document/d/1g9e-fFUAZz-sQgmLwQvD7wDdXK3E_0rwQQNjPQPCd6M/edit`

6 Analysis

Flaws:

1. One major flaw with this current implementation results from the fact that all Ethereum blocks are publicly inspectable. While this transparency does provide some benefits, especially when the Ethereum is supposed to be used as escrow, it is not ideal for exchanging private materials like keys or unencrypted data. However, the victim does not have much choice in a real ransomware attack anyway.

2. There is a potential "gas wasting attack" in the current implementation; this is when the victim, once given the address of the smart contract, can repeatedly call methods on the contract to drain its gas. This, in a sense, wastes the ransomer's ether. One way to counteract this is to have the victim create the smart contract. However, this makes the flow for the victim more complicated. In addition, the "gas wasting attack" may not

really be a significant issue, since, presumably, the victim's main priority is getting their key back.

3. Currently, the key that is sent back to the server is not encrypted. If the victim had a WireShark or some such device, they could easily read the request to the server for the key. Optimally, we would encrypt the key using RSA or other asymmetric encryption scheme with the ransomer's public key, as the encryption and decryption of the key could happen on the victim's and ransomer's own devices and do not bypass the smart contract (i.e. we would not be limited by the capabilities of the smart contract).

Limitations:

1. The current implementation requires all of the data to be sent to the smart contract. As a result, not that much data can be sent due to the limits of each block's processing power, and the cost of gas.

2. Other than being susceptible to analysis of the distribution of letters, the Caesar cipher we use only has a key space of 2^8, which is very easily brute-forced. However, given that not too much data can be encrypted at once, a one-time-pad would be more feasible to implement than in most cases, and would also be semantically secure.

7 Further Research

One possibility for handling the first flaw (about the blocks being publicly inspectable) is to verify that the ransomer provides the correct key with a zero-knowledge proof. Theoretically, this would make it so that, even though the blocks in Ethereum are publicly viewable, the keys and data themselves would not be revealed to any degree. Current research on introducing privacy and zero-knowledge into Ethereum, through integration with Zcash, is in progress, but limited by the computational power (gas) currently allowed for each block. [1]

However, zkSNARKs may not be necessary for specifically solving the problem involved with guaranteed ransomware. In our investigation, potential topics that could have helped us solve our problem were state channels, committment schemes, oblivious transfers, and secret sharing.

7.1 Proof of Correct Key

In the current implementation of our ransomware, we do a simple verification of the key by having the smart contract (a trusted party) decrypt the encrypted data that the victim inputs and confirm that the SHA256 hash produced on that data matches the hash provided by the victim. However, while finding a collision in SHA256 is difficult, we would ideally like to implement a more robust proof in order to give a more concrete guarantee to the victim. This would follow the idea of a zero-knowledge proof, and consist of the following steps.

1. Currently, the victim is instructed to provide one "data" input and one "hash" input. In a zero-knowledge proof, we would prompt the victim to input several different "data" and "hash" inputs for randomly selected blocks of their data. Ideally, this would be done automatically by the ransomware, although we cannot expect the victim to have the blockchain set up on their computer and thus must instruct the victim on how to interact with the smart contract through the GUI.

2. The ransomer would then need to provide a key to the smart contract (which is used for all decryptions—the attacker cannot change the key for each iteration of this procedure).

3. The smart contract decrypts the key and verifies the SHA256 hash matches.

4. he smart contract sends a message to the ransomer's server, indicating that the decryption was successful.

5. The victim is prompted to input the next set of "data" and "hash" bytes to the contract.

6. Once all the victim's input sets have decrypted properly, the smart contract delivers the key to the victim and the money to the ransomer.

With this proof, the victim can pretty much be guaranteed that the ransomer will only be paid if they provide the correct key, as it is practically impossible that the ransomer could find a key that would produce a hash collision for every randomly selected block of data the victim sends.

However, this imposes more limitations on the cipher we can use, as if a one-time pad were employed, the smart contract would need to know what portion of the one-time pad to use for the randomly selected chunk of the victim's data.

8 Challenges

In the final hours of completing this project, we were unable to implement a working smart contract that checked that the ransomer provided the correct key with the SHA256 hash. Well before the submission deadline, a number of elements of our submission were fully functional:

1. The client code was fully integrated with the server code, and could successfully send the key back to the ransomer's server (though unencrypted) and poll the server for the key. When the key was available, it would automatically decrypt the victim's data.

2. The server code was fully functional and able to interact with the smart contract.

3. The smart contract verification of payment was correct (payRansom)

However, we are unsure whether the "provideKey" code in the contract was functional. Testing on Remix (the online IDE for Solidity) proved to be unreliable. When running the payRansom method, it would consistently return false even though the amount paid was higher than the amount demanded. However, this function worked fine when run on the Ethereum test network. Thus, we could not determine whether the Remix IDE was outputting "false" because the hashes did not match (even when they should have matched) or because of the same glitch that caused it to return false even when it should have returned true in payRansom. When we ran this code on the test network, we found that the amount of memory we were using to check the hash (due to tedious type conversions between bytes32, bytes1[] memory, and bytes[] memory that made a simple Caesar cipher implementation take hours) caused the contract to cost an excessive amount of gas that exceeded the block limit. As a result, the integration of the overall system failed.

Given more time, we would perhaps try to implement another simple cipher that would not require iterating through each byte (again, a one-time pad would be great for this, as we could simply xor the entire bytes32 object with the one-time pad object and not need any conversions; however, as stated above, a one-time pad also comes with limitations like needing to know which section of the pad to xor with each chunk of input data). This would eliminate the need for type conversions that we were unable to understand, were immensely frustrating, and also cost a lot of memory and therefore an infeasible amount of gas.

9 Conclusion

Though the infrastructure to properly implement guaranteed ransomware is not present yet, this implementation demonstrates that the usage of a decentralized, impartial third party to resolve disputes such as that of guaranteed ransomware has been made much more viable with blockchain technologies.

As a group, we had a fantastic time exploring the theory and engineering behind Ethereum and the community around it. While we understand that there will be limited time to grade all of the projects, we would greatly appreciate any kind of feedback or suggestions for further research with regards to this project. Thank you!

References

[1] Christian Reitwiessner and Ariel Gabizon. An update on integrating zcash on ethereum (zoe), Jan 2017.