Charlie Dwyer, Carter Malecha, Max Fabian

Covert Channel Project Writeup

BACKGROUND ON COVERT CHANNELS

Covert channels were first coined as a term in about 1973, given a little wiggle room for when researchers tried to implement similar ideas a few years before. At first covert channels were at a very simple low level, like system load, or data transfer to determine information on a covert channel. The TCSEC (a set of computer security criteria) classifies 2 different types of covert channels, storage and timing. Storage relies on modifying a storage location to get information in a channel, and timing uses response time received to access information in a channel.

SIMILAR TOOLS

Our tool is relatively similar to other small decoder/encoder covert channel tools. It takes a message, breaks it down into something mostly undetectable, sends it to a receiver, and the opposite is done to accomplish the end message. Our tool uses binary conversion to set file size as our form of bit values. Other tools use binary conversions and time stamps to accomplish similar tasks and message sending techniques. System resource consumption as a covert channel was one of the first covert channels, and also one of the simplest, making it again similar to our tool. Our tool differs from other similar covert channel devices in the fact that it uses a temporary file directory to store each file, whether or not it has size (value) or not (no value). Some other covert channel tools use file formats like this but ours is a bit different than those specific tools.

HOW OUR TOOL WORKS

Here is a detailed breakdown of the functionality of the sender and receiver tools:

Sender: covert_channel_send_size

The sender tool encodes a message by converting it to binary and then creating a file sequence where the size of each file represents one bit of the message.

1. Converting Message to Binary
    - The message is converted into binary with the help of Python's "format(ord(c), '0b'). Each character in the message is converted into its 8 bit equivalent.
2. File Creation Representing Binary Digits
    - A file is then created for each bit in the binary message. The filename includes its timestamp to ensure the file has a unique name.
    - If the file's bit is '1', a piece of data is written into the file, making its file size greater than 0 bytes.
    - If the file's bit is '0', nothing is written into the file, keeping its file empty and size as 0 bytes.

Receiver: covert_channel_receive_size

The receiver tool decodes the sent message by looking at the sizes of the files in the directory, and translates it to its corresponding '0' or '1' binary digit.

1. Sort Files by Timestamp
    - The files in the received directory are sorted based on filename to ensure the files are read in the correct order they were created. This ensures that the original bit sequence is preserved.

2. Reconstruct Binary Message
    - The size of each file is checked:
        - If the file size is empty, a ‘0’ is appended to the ‘message_binary’ list
        - If the file size is greater than 0, a ‘1’ is appended to the ‘message_binary’ list
3. Decode Message
    - The binary digits stored in the ‘message_binary’ list are then converted back into a string by grouping the list into 8 bit segments and converting each group back to the corresponding character. The decoded message is then printed.

CREATION PROCESS

Here’s a breakdown of the process of creating our tool:

1. Brainstorming the Tool Idea
    - What we tried: Our main goal for the covert communication channel was to use the file system. We came up with the idea of using the size of files, where the data in the file represents binary digits ‘1’ or ‘0’. We wanted to send a message through generating files of different sizes depending on the content of the message.
    - What worked: Our core idea held pretty solid. By using file size as the mean to encode the binary data, we were able to effectively represent messages.
2. Implementing the Sender:
    - What was hard: One of the most challenging parts we found was figuring out how to correctly handle the timing of file creation. Initially, files were being written

too quickly which led to the pattern of the binary data being way off, so the message wasn't being translated correctly. We added a minor delay in the sender to fix this issue.

- What was easy: The binary encoding of the message being sent was super straightforward. Python's built-in functions helped us convert characters to binary super easily.

3. Implementing the Decoder
    - What was hard: We ran into some trouble with the behavior of the file system. Initially, we weren't sorting the files based on timestamp which created problems with reading the files in the correct order. We decided to name the files by their associated timestamp, so the files would be sorted in the correct order as they were created. Although this naming convention would lead us to be more cautious, as it would probably make the files more detectable.
    - What was easy: Once the files were correctly sorted, rebuilding the binary back to the original message is quite simple with Python's chr() function.
4. Overall Thoughts
    - What worked well: Our initial idea of using file size as the signal method for our covert channel was effective. This provided a clear and practical method to use the file system to hide messages
    - What didn't work: As mentioned above, making sure the files were created and read in the correct order was the trickiest part of the creation process. The file handling and delay management aspects needed some iterations.

- Challenges: Our main challenges of the covert channel was determining the balance for the timing of creation of the files, and correcting retrieving the message in the correct order.
- What was easy: Once our concept was clear, the actual sending and receiving process was quick and simple because of Python's easy to use file I/O and string manipulation capabilities.

DETECTION AND OBSERVABILITY

One of the major ways that this can be detected is by viewing all of these random files that are seemingly almost empty. Even though the information within the files won't raise any suspicion, seeing a bunch of random files in a folder could likely raise some red flags. If someone were to notice the files, they may likely just get deleted as all the information is really in the file size and not in the information that is written inside them.

Another area that could lead to potential detection is by watching the network traffic. If someone is watching for the messages and suspects that something is going on, it is possible that it will be noticed. However, using a secure network will make this much harder for someone to notice that a covert channel is being used to send information.

Overall, this covert channel is difficult to detect unless you are actively looking for it. It is highly unlikely you would be digging through files and stumble across a bunch of empty or almost empty files. It is also highly unlikely one would be watching network traffic to find the messages. As long as someone does not suspect that anything is going on, there would be no reason to be looking for a covert channel and no reason for someone to find this one.