Technion, Winter 2011/12 Semester

236349 - Project in Computer Security

Attacking Tor with covert channel based on cell counting

Final report

Matan Peled	200729218
Dan Goldstein	052861747
Alexander Yavorovsky	318109477

Project was made under the guidance of **Amichai Shulman**.

Overview

Tor is a circuit-based low-latency anonymous communication service. It tries to provide a reasonable balance between anonymity, usability and efficiency [1]. Playing the role of web server administrators we have devised a technique to reveal the true identities of clients who use Tor to access our website. Our idea relies on the low-latency characteristics of Tor routing and on Cell counting (Cell being the basic unit of information that Tor works with).

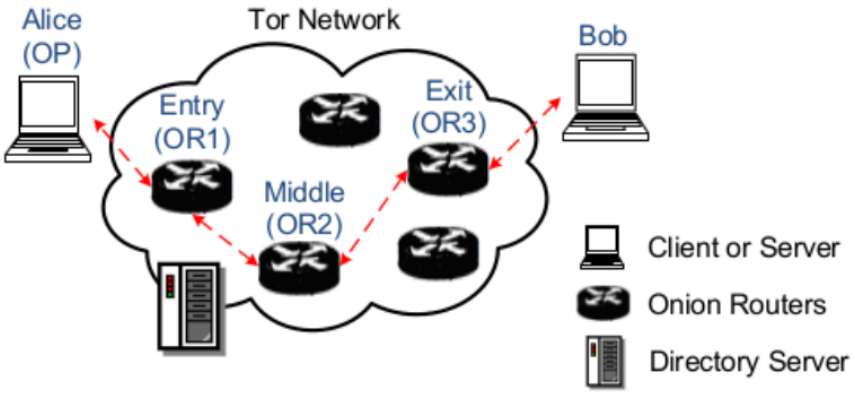
Tor

Tor is composed of several elements.

- The Tor network. The Tor network is composed of volunteers computers that have a Tor program installed on them and chose to allow the computers to be used as 'Tor relays' . These Tor relays are used to construct circuits through which the clients traffic is bounced before reaching the original destination.
- Directory servers. These are centralized servers that are globally known to all Tor clients, and provide a consensus list of all available Tor relay every hour. The purpose of the directory servers is to provide clients of Tor with the IP addresses of Tor relays in order for the clients to be able to construct the circuits through which their traffic is going to be routed through. It is worth mentioning that because clients always need to consult with directory servers in order to able to later construct the circuits through which their traffic would be routed, it is extremely easy to reveal the identities of Tor users. That being said, lately Tor website began offering alternative ways to access the Tor network, using Tor bridges - which are not supplied through a centralized server.
- Exit nodes. These are Tor relays that may be used as the last hop the clients traffic goes through before reaching its destination. Usually Exit nodes configure their policy to limit the kind of traffic that is allowed to go out of them in order to prevent abuse of the Tor network by criminals, spammers and other immoral clients.
- Entry nodes - Tor relays that pose as the first hop, the first relay that clients of Tor send their traffic to.
- Clients - people who use Tor. In technical terms, the piece of software that is installed on their computers is called OP - Onion Proxy. Sending all of the network traffic through the Onion Proxy makes it go through the Tor network, providing anonymity to the users.

When Tor clients wish to access the internet in order to surf the web (for example) the OP performs the following tasks: First it has to construct a circuit through which the clients traffic is going to be routed through. In order to do that, OP contacts the

Directory servers, requesting a list of the available Tor relays. It is worth mentioning that the Directory servers provide the same list of relays to all clients - which is critical to anonymity since if different clients got different lists, by differentiating the lists one could extract some details about the user's identity if not compromise it entirely. After OP has all the list of the relays it tries to construct a pseudo random circuit through them while trying to ensure that the circuit is reasonably low latency. The construction of the circuit involves negotiating a symmetric cryptographic key with each of the Tor relays selected to be in the path (by default there are 3 Tor relays in a circuit - entry, middle and exit).



[2]

Once the circuit building is complete the OP can start sending the clients messages through the Tor network. Before sending the message, OP encrypts it in onion-like fashion: first the message is encrypted with the Exit nodes key, then this encrypted message is encrypted again with Middle node key (middle node is the mid-way node between the entry and exit nodes) and then finally it is encrypted yet again with Entry nodes key.

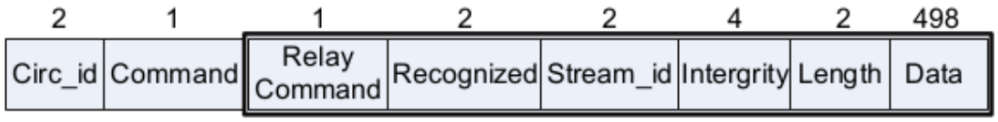
When receiving the message, the Entry node decrypts the first layer of encryption, revealing the next destination along the circuit - the identity of the middle node. Same goes for the middle node and when the message reaches the Exit node it decrypts the last layer of encryption (meaning the original message, in case it is not encrypted before being sent through Tor, can be read by the Exit node) and sends it to the original destination.

The encryption mechanism ensures that each Tor relay knows only the previous and next hops along the circuit - never seeing the whole circuit. Thus, the entry node knows the identity of the Tor client but it doesn't know the where the requests are being sent to, and the Exit node knows the identity of the destination but not that of the client. So, in order to be able to break the anonymity of Tor users an eavesdropper has to be able to know the circuit the client is using and conduct traffic analysis of the traffic that goes through all of the 3 (or more) Tor relays along the circuit. That is nearly impossible, even for intelligence agencies as Tor makes a point of trying to construct the circuit in such a way that each Tor relay is in a different continent or at

least in a different country. Additionally, Tor changes the circuit every once in a while - making the task of determining the routing circuit even harder.

Even though Tor introduces a considerable delays to the traffic being routed through it (establishing symmetric keys, encrypting and decrypting, etc) it still is reasonably low-latency, low enough even for VOIP to work occasionally (as different built circuits have varying latency).

How Tor handles Data

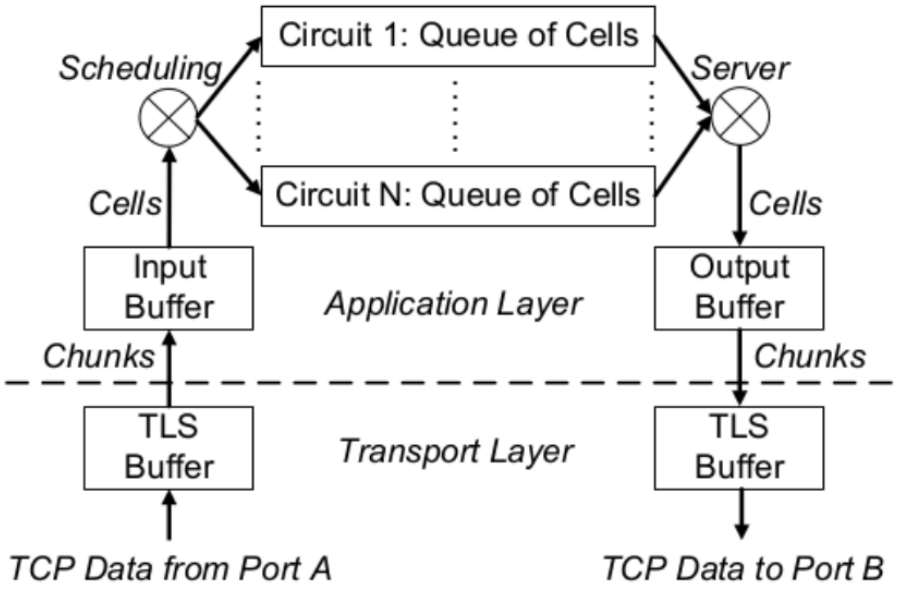


Tor Realy Cell Format [2]

Tors basic unit of information is called a Cell. This unit includes a 498 bytes of user data and other information relevant to Tor such as Relay commands and so on. Conceptually, Tor sits in the application layer and is using the TLS layer for encryption and secure communication. Data coming from the TLS layer is put in the Tor input buffer and data going from the Tor layer into the TLS (the Cells that are being sent) goes into the output buffer and then to the TLS layer. There are two distinct “events” that Tor uses to send and receive data - the write and read events.

Once the read event is called, cells are pulled from the TLS buffer into the input buffer at the Tor application layer. The chunks of data that are being pulled from TLS are divided into Cells and are stored in this format in all of the Tor buffers. Once in the input buffer, one cell (the first one) can be pulled from the input buffer into the circuit queue. There multiple queues, one per circuit that is going through the Tor application. In the occasion that the output buffer is empty the cell can now be put there (only the first one). Other cells will have to be read from the input buffer into the queue and wait for the write event that flushes the output buffer before being put in the output buffer.

When the write event is called all of the cells in the output buffer are flushed into the TLS layer (sent away to their respective destinations) and then the contents of all of the queues can be put in the output buffer - to await the next write event in order to be sent.



Processing the Cells at Onion Routers [2]

This mechanism of Cell handling that Tor employs means that when one sends data that is big enough to fit into 3 cells (for example 3*498 bytes fits perfectly) it is reasonable to assume that the order of arrival of this data at the destination is going to be: a packet that contains 1 cell, and then a packet that contains 2 cells. This, however, cannot be extended to more than 4 cells because at 5 cells the second packets (that is supposed to contain 4 cells) almost always is split into 2 packets by the IP layer due to the MTU limitations of the communication lines along the circuit.

Additionally, the scenarios we have described thus far are overly simplistic and ignore various effects of lag and load on the Tor network. For example, in the case that our circuit goes through a very busy Tor relay (or one that has bad connectivity) and the source sends 3 cells of data they may arrive in 3 separate packets, instead of 2 as we have described above. The occurrences of such patterns is not frequent and can be written off as anomalies that happen from time to time that may lead to inaccurate analysis but in most cases can be safely ignored.

The Attack

First - the assumptions that we make in order for the attack to work successfully.

- The attacker has control of the web server or can modulate the outgoing traffic from the web server.

- The attacker has control over the entry node that the client is using or can sniff the traffic that goes from the entry node to the client.

The scenario we are dealing with: we are the webmasters of a web server and have clients that access our web site through the Tor network. Our goal is to identify the real IP addresses which these clients are using.

The Server

Because our attack partially relies on inter packet delays we have disabled the Nagle's algorithm at the TCP layer. Nagles algorithm buffer data from the application layer, at the transport layer waiting for a reasonable chunk of data to form before sending it off.

When our web server receives a GET request it generates a unique client id. This id can consists of any number of bits we choose to - our default implementation uses id length of 12. Then this id is encoded with the Reed Solomon error correction code (the parameters for this algorithm can be configured easily - allowing for any number of extra error correcting bits to be added) and translated into hexadecimal digits. These hex digits are then encoded into the data stream of the outgoing server response. An example will clarify this:
Suppose that after the Reed Solomon error correction encoding is done with the generated id the string we have is 0x1A4. The way the server encodes this id into cells is it sends a chunk of 1*498 bytes of data, then waits for 0.5 seconds (the inter-send delay interval can be reconfigured) , then sends a chunk of 10*498 bytes and then waits again for 0.5 seconds and then finally sends a chunk of 4*498 bytes of data.

The reason for sending the data in this fashion is that Tor packs the data sent through it into cells, each cell consisting of 498 of data. So, when we send 498 bytes of data through Tor, Tor puts this data in a single Cell. Using this we control exactly how much cells we send through Tor, and by introducing the delay between each burst of cells we allow traffic analysis on the entry node to detect and distinguish the bursts from each other - allowing us to count how many cells were sent in each burst.

To summarize the work flow of the server:

- Generate unique client id per request
- encode the id with Reed solomon
- Interpret the encoded id as digits in whatever base we want to use (hex, binary - the base doesn't mean anything because we can encode any number into the cell count)
- for each digit in the interpreted id:
    - send a burst of [digit] number of cells (498*[digit] bytes)
    - wait 0.5 seconds

Assuming we encode a string of x digits we introduce a total delay of x*inter-burst-

delay seconds. For example using 16bits encoded as hex digits and the inter-burst interval set to 0.5 seconds introduces a delay of 2 seconds. This isn't bad at all, considered the fact that surfing through Tor users often experience delays of 5 seconds or even more.

One last thing to mention here is that in addition to the cell bursts the server also sends out a "marker" burst. This marker is going to be used later, in the traffic analysis of the entry node traffic to identify the streams that belong to clients of our web server (since an entry node can be used by many Tor clients besides the ones that access our website) and mark the beginning of the encoded cell bursts so that the decoding code can determine from where to start counting the cells. This "marker" doesn't have any impact on the encoded id nor it is affected by the id. However, it does introduce additional delay to the stream.

Entry node and decryption

The entry node is set to run tcpdump at all times - recording all of the traffic that goes through it. At anytime we decide we want to see what clients have used the entry node to surf our website we take the packet capture file that tcpdump generated and run our decryption program on it.

The decryption program analyzes the packet capture file for outgoing traffic only (from the entry node to hosts) and using "tshark" program it generates a separate file per each unique IP that appears in the packet capture file. After this stage is done, each IP packet file is analyzed and searched for the unique marker that we embed in the outgoing traffic from our web server. In the case that the marker isn't found we determine that the IP being analyzed doesn't belong to a client that tried to access our website but rather to another user of the Tor network that uses our Entry node. In the case that the marker if found, the decoding process begins - the program analyzes each packet that comes after the end of the marker for its size and tries to determine the amount of cells that are contained in the packet. The exact formula is derived as follows:

TCP header = 32
IP header = 20
EMPTY TLS Record = 5+20+12 = 37
TLS Cell = 512 (size of a cell) + 5 + 32

From this we can derive that the total number of cells contained in a packet of size X bytes is:

$$\frac{X - (20 + 32)}{2*(5 + 20 + 12) + 512}$$

Sometimes this calculation doesn't give us an integer, meaning that some cell was split between two packets - but our calculation isn't affected by this because we don't round off until we sum up all of the cell counts of all the packets in the current burst.

The counting of cells stops when the following conditions hold true:

- The delay the program sees between the last packet and following is greater then some constant we found to be consistent with the end of a burst. This is the reason we have inter burst delay interval and altering the 0.5 parameter will have an effect on how we try to detect the start and end of a burst.
- There is exactly one cell in the following packet.

After we find the end of the burst we sum all of the cell counts and round off any accumilitive error there might have happened along the way. The result is the encoded digit the server has sent. We append this digit to the decoded id and carry on to analyze the next burst.

This process stops once the decoding program has accumulated the exact number of expected digits in the decoded id (it knows how many bits there are in an id).

Since the decoding process often takes a lot of time (depending on the packet capture file size and how many distinct IPs were recorded in it), we have implemented a parallel version that decodes each unique IP file in parallel to others. This can be used/ invoked with passing the "-p" option to the decoding program.

The Client

The client code is very simple - it establishes a connection through Tor to the server and sends a request, and then after receiving the data exits.

Results

Planned Tests

We planned to run experiments with the setup mentioned earlier - modifying the inter cell batch sleep time interval and marker sleep interval. To remind, inter cell batch interval is how long the server pauses and sleeps between encoding each digit (that is encoding via a batch of cells). The marker sleep interval is how long the server sleeps between sending the cells of the marker. We have found that using a marker of less than 2 cells makes it very difficult to detect where our encoding starts and thus damages successful decoding rates significantly, so we stuck with marker of length of 3 cells with marker interval of sleep time between each of them. In order for the communication to appear as delay-less as possible narrowing down this interval is important. We planned to experiment with delays of 3, 1.5, 1 and 0.5 seconds.

Additionally, the other parameter that is important for minimizing introduced delay is the inter-batch sleep time. We planned to experiment with delays of 1, 0.75, 0.5 and 0.25 seconds.

For each combination of the mentioned delays, we planned to try 100 connections in total over the span of several days in order to reduce the significance of any irregular negative effects like heavy loads on the tor network or bad choices of Tor relays might have on our results.

The Results

Unfortunately, the idea that we eventually found to be working best came at a very late stage and thus we didn't have enough time to do proper experiments, although we did plan a major

part of them.

We can only offer what we saw from hours of experimentation with the code and other various parameters - during our attempts to optimize the attack implementation. Eventually, we have arrived at the conclusion that marker intervals of 1.25 seconds and inter batch intervals of 0.5 seconds offer the best successful decoding rate against the lowest delay tradeoff. We were able to successfully decode roughly 80 percent (probably more) of all the encoded client IDs. It is important to notice that because the client will likely send more than one request to our webserver and because we encode an id when replying to each request the probability that we will be able to successfully identify the identity of the client is significantly larger than 80 percent because the random variable that expresses the number of attempts needed in order to successfully decode his id is geometrically distributed with success parameter of p=0.8.

Countermeasures

Because Tor strives to provide anonymity with low latency it doesn't force and introduce fake delays between the packets sent through the circuit. In case there was a random delay inserted between each two packets sent through Tor it could potentially throw off the decoding algorithm, which distinguishes subsequent bursts by the delay between last and first packets of each burst.

Another thing Tor could do to prevent this attack is to randomly pad data sent through it. So, if for example the data sent is 498 bytes in size and tor splits the data into two padded cells our decryption algorithm wouldn't be able to tell that only a single cell was originally sent. This padding can be scaled to any number of cells - and if kept to a minimum it shouldn't affect too much the latency of connections. Additionally, this dummy data could be randomly dropped in the intermediate routers in order to further confuse attackers.

Third option is for Tor to use Cells with varying data size field. If the amount of data contained in a cell isn't constant it would require deep packet inspection to see exactly how the cells are structured instead of relying just on the size of packets to figure out how many cells are contained in the packet. And since the traffic is encrypted we can't do that - there is no way to look into how to cell looks like at the entry node.

Other approaches that failed

Before arriving at the final implementation of the attack we had some ideas along the way that have failed to work. This short passage describes some of them.

Originally, we started of with the idea that we can encode the id of clients in the inter packet delays. While conceptually this idea is sound as Tor uses low latency circuits and thus shouldn't affect too much the introduced inter packet delays, in practice this

approach fails miserably. At resolution of 0.5 seconds (meaning we encoded digit 0 with 0.5 seconds delay, digit 1 with 1 seconds delay and so on) this method works quite well but the client suffers from unreasonable delay until his request is replied. Trying to lower the resolution makes the inter-packet delays incomprehensible, and we couldn't decode any significant portion of ids. Later, when we delved deeper into the inner workings of Tor the reason behind the failure became clearer as Tor sometimes merges cells together and sends them in one packet, depending on the delay between them. When delays are shorter (say 0.1) merges happen often and so we lose bits of information sent and worse yet - we can't distinguish the intervals one from another. We tried to augment this approach by encoding the id with the Reed solomon error correcting code but this didn't help much if at all.

Another idea we had was to try and encode binary digits by sending big chunks of data to signal 0 and introducing a delay to signal a 1. So, for example - if the id was 10010 we would send a small packet (on cell in size) then send a big chunk of data followed by another and then delay between sending a small chunk, after which we would send the final big chunk. When decoding, the big chunks of data would be detected by seeing a lot of packets arriving at very close timings and with very big packet contents in size. The delays would be detected because we ensure that the delay is big enough so it wouldn't be lost through the circuit (delays above 0.75 worked very well).
This approach worked pretty well but we decided to abandon it since it required large chunks of data and in case we don't have much data to send there would be a lot of padding.

Suggested improvements

Instead of writing a custom server, the implementation should be in a reverse proxy form. That is, when a client wishes to connect to our server it instead connects to the reverse proxy which connects to the server on his behalf and then buffer the servers response - modulating the response in any way we want. This would prevent us from having to change any server code and make our implementation much easier to use.

The decoding code could be much improved. This requires additional extensive testing but could be done automatically if some kind of automatic learning process was employed. There are artificial intelligence algorithms that build a classifier based on sample data set - an example would be a classifier that learns to distinguish between dogs and non-dogs based on a data set that consists of dogs and other living creatures. In order to create the training data set, we could create thousands of connections sending a generated id in each and then classifying the results automatically as we already know what ids we have sent. The only part that we wouldn't be able to perform automatically is creating a set of "properties" or "attributes" according to which the classification would work.

Conclusions

From our impressions during tuning of the attack code we have concluded that the attack works well. However, to our dismay the lack of time for serious testing didn't allow us to show any real statistics that support this claim.

There is need for further improvements such as creating a learning agent that will be able to identify cell batches with greater accuracy - allowing us to reduce the delays the attack causes.

References

[1] - Tor: The Second-Generation Onion Router (Roger Dingledine, Nick Mathewson, Paul Syverson)
[2] - Equal-sized Cells Mean Equal-sized Packets in Tor? (Zhen Ling ,Junzhou Luo , Wei Yu and Xinwen Fu)
[3] - A New Cell-Counting-Based Attack Against Tor (Zhen Ling, Junzhou Luo, Member, IEEE, Wei Yu, Xinwen Fu, Dong Xuan, and Weijia Jia)