Lachlan Kang
Efficient botnet herding within the Tor network
Journal of Computer Virology and Hacking Techniques, 2014; OnlinePubl:1-8

myjournal manuscript No.
(will be inserted by the editor)

Efficient botnet herding within the Tor network

Lachlan Kang

Received: date / Accepted: date

Abstract During 2013 the Tor network had a massive spike in new users as a botnet started using Tor hidden services to hide its C&C (Command and Control) servers. This resulted in network congestion and reduced performance for all users. Tor hidden services are attractive to botnet herders because they provide anonymity for both the C&C servers and the bots. The aim of this paper is to present a superior way that Tor hidden services can be used for botnet C&C which minimises harm to the Tor network while retaining all security benefits.

Keywords Tor hidden services · Botnet command & control · Botnet architecture · Tor security

1 Introduction

Tor is a public anonymity network that runs on free open-source software developed by the Tor Project [11]. In broad terms an anonymity network is a network that you can forward your Internet traffic over in order to hide your IP address from end servers. Tor hidden services are feature of Tor that allow you to anonymise both ends of a connection: the client doesn't know the identity (IP address) of the server, and the server doesn't know the identity of the client.

In late 2013 the Tor network suddenly came under heavy load due to a botnet that had begun using Tor hidden services to hide its C&C (Command & Control) servers [1,3]. The botnet using Tor was initially dubbed Mevade [3], however, it was later discovered that Mevade was an existing botnet called Sefnit that had been upgraded to use Tor [5].

With regards to botnets, C&C refers to the management of a botnet by its *herder*, who is typically its owner. A C&C scheme is a method of propagating orders from a botnet herder to the hundreds, thousands, or even millions of bots in their botnet. Botnet orders can be all sorts of things; for instance some botnets are used for click-fraud [2], others are used to perform DDoS attacks [7], and some are used to send email spam [9]. The purpose of the Sefnit botnet was to commit click-fraud and mine Bitcoin [3,5,6].

Since Sefnit appeared in the Tor network the number of bots has been slowly decreasing as efforts have been made by Microsoft [6] and others to shut it down. Later versions of the Sefnit software were reported to no longer use Tor [8]; instead they switched back to an SSH based scheme which had been used prior to Tor. Tor is now down from approximately 5.5 million users (4.5 million suspected bots) to approximately 2.5 million users [10].

We provide here a relatively efficient scheme for performing botnet C&C using features of Tor hidden services. The goal here is to allow a botnet to coexist peacefully with regular Tor clients in the network.
The use of Tor provides several potential benefits to a C&C scheme:

- The Tor network is a public network with hundreds of thousands of users. Bots are able to hide among those legitimate users to prevent identification. This is a large benefit over creating and using a private bot-only network.
- Uploads and downloads of orders can be anonymised. This is true in that the uploader and downloader IP addresses are kept secret; taking control over a single bot or server doesn't

L. Kang
The University of Adelaide
E-mail: phvmfgw7@gmail.com

allow you to track down other servers or bots. It's also true in that there is no registration or use of user accounts.

Additionally our particular use Tor provides an extra benefit:

– Uploaded orders are hosted by neutral third-party (non-bot) servers who don't know what they're serving. This is great because it doesn't require bots reveal their identity and publicly serve orders to other bots.

Compared to other C&C schemes that make use of Tor hidden services, for instance an IRC server accessible as a hidden service, ours is far more efficient, less damaging to the network and more scalable. On the down side it's less responsive than an IRC based system. Orders are *pulled* by bots periodically rather than *pushed* from the herder out to the bots. The responsiveness can be anything; it's a trade-off between frequency of bots pulling orders and the overall negative effect that the botnet has on the Tor network. Our scheme is also potentially more secure versus deanonymisation, as it only utilizes the first half of the Tor hidden services protocol. Since it's far simpler than a scheme that uses the entirety of the hidden services protocol there is less avenue for attack.

Our scheme works by making use of the name-resolution system that's built into the Tor hidden services protocol. The name-resolution system is used to resolve a hidden service's `.onion` address into information that is required to connect to that hidden service. The name resolution system consists of a DHT (Distributed Hash Table) which hidden service descriptors are uploaded to. A hidden service descriptor is a document that contains, primarily, a list of the hidden service's introduction points. A hidden service's introduction points are onion-routers which can be contacted by a client to establish a connection to that hidden service.

In Tor hidden services V2 [12] the list of introduction points can be optionally encrypted. In Tor hidden services V3 [4] (currently unimplemented and in its draft phase) the list of introduction points is always encrypted. Because this list is encrypted the servers in the DHT that host the descriptors are unable to see its real contents. This means that the servers are unable to validate the contents of the encrypted section, and hence we can include arbitrary data instead of a list of introduction points.

In this paper we propose a method of botnet C&C that uses the encrypted section of hidden service descriptors to store and propagate orders for bots. Because V3 hidden services are currently unimplemented we will only describe the scheme in terms of V2 hidden services. In Section 2 we discuss the format of hidden service descriptors, in Section 3 we discuss how we make use of descriptors in our scheme, and in Section 4 we discuss how descriptors are published and downloaded. In Section 5 we then discuss how to avoid damaging the Tor network, and then in Section 6 we discuss potential countermeasures that the Tor project could use against a botnet implementing our scheme.

2 Descriptor format

The benefit of using Tor to a botnet is primarily anonymity; the same benefit provided to regular hidden service operators and clients. Hidden services are designed to anonymise both ends of a connection, the client and the server. Hidden service descriptors must be uploaded anonymously, but must also be downloadable and usable by any client who knows the hidden service's `.onion` address. Hence, if we use hidden service descriptors to contain C&C information botnet herders will be able to publish commands anonymously and clients will be able to download them anonymously.

A `.onion` address is used to identify a hidden service in the network. It is calculated based on the hidden service's long-term-public-identity-key and never changes:

$$\texttt{onion\_address} = \text{base32}(\texttt{permanent\_id}) \;||\; \text{“.onion”},$$

where

$$\texttt{permanent\_id} = \text{substr}(0, 20, \text{hex}(\text{SHA1}(\text{substr}(22, \text{DER}(\texttt{public\_identity\_key}))))),$$

the hidden service's long-term-public-identity key encoded in DER, with the first 22 bytes cut off, hashed using SHA1 and truncated to the first 80 bits.

From the hidden service's `.onion` address (remember that this contains `permanent_id`) and the current time we can then calculate *descriptor-ids*. Descriptor-ids are used to uniquely identify each hidden service descriptor in such a way that the associated `.onion` address is kept secret.

$$\texttt{descriptor\_id} = \text{SHA1}(\text{bytes}(\texttt{permanent\_id}) \;||\; \text{bytes}(\texttt{secret\_id\_part})),$$

where

$$\texttt{secret\_id\_part} = \text{SHA1}(\text{bytes}(\texttt{time\_period}) \;||\; \text{byte}(\texttt{replica})),$$

and

$$\begin{aligned}
\texttt{replica} &= \text{An integer.}\\
&\quad\ \text{Either 0 or 1 in regular Tor.}\\
&\quad\ \text{We use this as our page number.}\\
\texttt{time_period} &= (\texttt{current_time} +\\
&\quad\ \texttt{permanent_id_byte} *\\
&\quad\ 86400/256)/86400,\\
\texttt{current_time} &= \text{The number of seconds}\\
&\quad\ \text{since 1970-01-01 00:00,}\\
\texttt{permanent_id_byte} &= \texttt{permanent_id[0]}\\
&= \text{The first byte of } \texttt{permanent_id}.
\end{aligned}$$

V2 hidden service descriptors take the following form [12]:

```
rendezvous-service-descriptor <descriptor_id>
version 2
permanent-key
-----BEGIN RSA PUBLIC KEY-----
<public_identity_key>
-----END RSA PUBLIC KEY-----
secret-id-part <secret_id_part>
publication-time <publication_time>
protocol-versions 2,3
introduction-points
-----BEGIN MESSAGE-----
<encrypted_data>
-----END MESSAGE-----
signature
-----BEGIN SIGNATURE-----
<descriptor_signature>
-----END SIGNATURE-----
```

Where the value `<descriptor_signature>` is the bytes of the SHA1 hash of everything before (and including) the text `signature`, encrypted using the hidden service's private identity key (RSA private-key encryption using PKCS.1 padding).

Where `<encrypted_data>` is where we put our C&C data. Usually this contains a list of introduction points which may or may not be encrypted. Hidden Service Directories (HSDirs), the onion-routers that host hidden service descriptors, treat this as if it's always encrypted and hence are unable to validate the section because they have no way to decrypt its contents; this allows us to include arbitrary data instead, as little to no validation is performed on it. Section 3 describes how we use this section in our scheme.

The contents of the `<encrypted_data>` section are quite flexible. For instance the contents do *not* need to be encoded in base64: it is willing to accept strings with non-base64 characters such as spaces, however, some characters such as quotes will be rejected. The contents don't actually need to be encrypted either. Lines do not need to be wrapped to 64 characters. There is no minimum length. It will accept 0 characters between the "`-----BEGIN MESSAGE-----`" and "`-----END MESSAGE-----`" tags. And the maximum length appears to be 10,000 characters.

The value of `<publication_time>` is the time that the descriptor was published in the format `<YYYY>-<MM>-<DD> <HH>`:00:00 (we round the time down to the hour [12] so we set the minutes and seconds to 00).[1]

And where the remaining values between angle brackets are place holders for values that we used earlier in the section:

$$\begin{aligned}
\texttt{<descriptor_id>} &= \text{base32}(\texttt{descriptor_id}),\\
\texttt{<public_identity_key>} &= \text{PEM}(\texttt{public_identity_key}),\\
\texttt{<secret_id_part>} &= \text{base32}(\texttt{secret_id_part}).
\end{aligned}$$

3 Architecture

In regular Tor each active `.onion` address has, at any time, two hidden service descriptors, each hosted at three servers, called Hidden Service Directories (HSDirs). Six servers is fine for small botnets, but when we have several million bots all downloading the current set of commands it is likely to be far too little.

To improve scalability of the botnet's C&C architecture we use a simple two-level hierarchical topology for the servers. The bots all come with a hard-coded a set of public-keys that we'll refer to as *root public-keys*. From each of these keys a `.onion` address can be derived (as described in Section 2). We'll refer to these `.onion` addresses as root `.onion` addresses. The descriptors that can be download using a root `.onion` address all contain a list of other `.onion` addresses; those addresses are each called intermediate `.onion` addresses.

The descriptors that can be downloaded using an intermediate `.onion` address each contain a copy of the current orders from the botnet herder. All intermediate `.onion` addresses are used to host the same set of orders.

$$\begin{aligned}
\text{root_onion_address}_i &\xrightarrow{fetches} \text{List of all}\\
&\quad\ \text{intermediate onion addresses.}\\
\text{intermediate_onion_address}_j &\xrightarrow{fetches} j\text{'th copy of the}\\
&\quad\ \text{current botnet orders.}
\end{aligned}$$

[1] While the specification [12] states we should round down to the nearest hour it is not always done in practice.

When a new bot enters the network it must bootstrap itself. It does so by randomly selecting one of the root `.onion` address, then using that it downloads the list of intermediate `.onion` addresses. This process closely mirrors the order fetching process shown in Section 4; the main difference being that a root `.onion` address is used instead of an intermediate one. The `<public_identity_key>` in this descriptor must match one of the root public keys that the bot has hard-coded, otherwise the descriptor is considered a forgery[2]. From then onward the bot periodically downloads new sets of orders, each time using a randomly selected intermediate `.onion` address. Period lengths could be every hour, every day, or anything else. It depends on how responsive you want the botnet to be; a short period will make the botnet more responsive, however it may overload the HSDirs with too many requests.

Bots should select their initial download time at random. Say this isn't done and instead the download-period was 24 hours with each bot making their first download at 2:15am; the intermediate servers would be hit with thousands of requests at 2:15am each day but none at other times. This sudden spike in traffic would appear quite suspicious, and, if enough bots were in the botnet (and there were too few intermediate servers) it might actually cause the intermediate servers to be DDoS'd. If each bot selects a random time for their first download, requests would be more evenly distributed over each download period and the traffic would seem more normal.

Currently the maximum amount of data we can squeeze into a single descriptor is 10,000 characters. We can get around this character limit by making use of replica numbers. In regular Tor every descriptor has two replicas: replica 0 and replica 1. They are both exactly the same except for the values that their replica number modifies: `secret_id_part` and `descriptor_id`, as well as the descriptor signature. Replica numbers are not included in hidden service descriptors so there is no way for a HSDir to determine if a replica number other than 0 or 1 is actually used. This allows us to re-purpose the replica number, and instead use it as a page number. For example, the first page of our botnet orders is page 0, and hence its replica number is 0. If the orders can't be squeezed into a single hidden service descriptor we put a "is last page?" marker inside page 0 that specifies that this is not the last page of orders. We then publish another page of orders: page 1, whose replica number is 1. If a third page of orders is required we do the same thing over again; we put a "is last page?" marker inside page 1 that specifies it's not the last page, then publish a third page of orders, page 2 whose replica number is 2. This goes on and on until the entirety of the orders are published. The last page would then contain an "is last page?" marker specifying that it is indeed the last page.

Figure 1 demonstrates how this page system works.

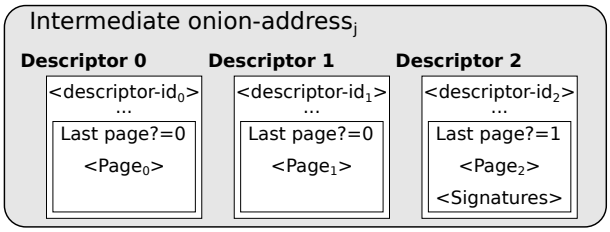

Fig. 1 The jth copy of a three page set of orders. Each descriptor is hosted by 3 different servers (HSDirs). Our orders are the concatenation of all three pages. All three descriptors are created using the same `.onion` address, and Descriptor i is the descriptor whose `secret_id_part` and `descriptor_id` were calculated using the replica number i. The inner box represents the `<encrypted_data>` section of the descriptor. The Signatures are each the hash all pages and all descriptor-ids, signed using one of the private root key; this proves that the orders are legitimate and prevents hostile takeover of the botnet.

As the size of the botnet increases we can add more intermediate `.onion` addresses to create more mirrors for the orders and hence reduce the burden on the existing intermediate servers. Every intermediate `.onion` address increases the number of copies of the orders being hosted by 3. The number of root `.onion` addresses never changes, however, but since bots only contact them once during their lifetime their scalability isn't a huge issue.

Sets of orders and the lists of intermediate `.onion` addresses should be re-uploaded by the herder every hour or whenever their content changes. This is to keep the orders current and to prevent expiration of orders.

All lists of intermediate `.onion` addresses and all sets of orders should be signed using all the root private-keys whose associated public keys were used to generate the root `.onion` addresses. There should be one signature for each root public-key. A signature is created by concatenating all the pages and `descriptor-id`s together, in order, then encrypting that hash using one of the root private-keys. This makes hijacking of the botnet significantly harder as it requires a herder to know a whole set of private keys rather than a single one. We include the `descriptor-id`s in the hash because doing

[2] Note that when they're implemented, V3 hidden services won't require this step as an `.onion` address will consist of a hidden service's *whole* public identity key encoded in base32. This means there will be no danger of hash collisions as there is with the current hidden service address scheme.

so ties the pages to the descriptors, and hence prevents replay attacks from an attacker who knows one or more of the root private-keys. Alternatively you could introduce another key pair, an order-signing key pair that is used solely for signing orders; only the herder would know the private key, and they would use it to sign orders instead of signing using all the root private-keys.

The following formula shows how a set of signatures should be created for a set of descriptors hosted under the same `.onion` address. The function hash can be any hash function, and the function Encrypt_i encrypts the hash using the ith root private-key.

$$\begin{aligned}\text{Signatures} &= \{\text{Signature}_0,\ \ldots,\ \text{Signature}_k\}\\ \text{Signature}_i &= \text{Encrypt}_i(\\ &\qquad \text{hash}(\texttt{descriptor-id}_0 \mid\mid \texttt{page}_0\\ &\qquad\quad \mid\mid \ldots\\ &\qquad\quad \mid\mid \texttt{descriptor-id}_k \mid\mid \texttt{page}_k))\end{aligned}$$

Figure 1 demonstrates where the list of signatures should be included: along with the last page of orders.

4 Publishing and fetching orders

When a bot wants to fetch the latest set of orders it does as follows:

1. The bot selects one of the intermediate `.onion` addresses at random. It knows these from when it bootstrapped itself after its initial infection.
2. It calculates the `descriptor-id` of the first page using the selected intermediate `.onion` address and the replica number 0.
3. The bot downloads the descriptor with that `descriptor-id`.
   (a) If the bot isn't able to download that descriptor it should wait for a short time, then start again at Step 1.
   (b) If the bot successfully downloads the descriptor it should continue to Step 4
4. The bot checks that the descriptor is a valid descriptor. This involves parsing the descriptor and checking that each field is valid (Section 2).
   (a) If the descriptor fails validation the bot should go back to Step 1.
   (b) If the descriptor is valid the bot should continue to Step 5.
5. The bot extracts the page from the descriptor and checks to see if it's the last page.
   (a) If it's not the last page, the bot calculates the `descriptor-id` of the next page by incrementing the replica number, then proceeds to Step 3.
   (b) If it is the last page the bot proceeds to Step 6.
6. The bot now has all pages and validates them by checking that the signatures on the last page are correct. Section 3 describes what these signatures are and how they're created. The signatures are validated by decrypting each hash using the root public-keys, then recalculating the hash and comparing it to the decrypted values.
   (a) If the decrypted hashes don't match the calculated one the pages are invalid and the bot should start again at Step 1.
   (b) If the decrypted hashes do match then the pages are valid and the orders can be obtained by concatenating all pages together in order.

The client fetches the first page of the current orders from a randomly selected intermediate server. The client then continues to fetch subsequent descriptors (pages) by increasing the replica number and downloading descriptors until finally they reach the last page.

Note that the easiest way to perform Steps 2, 3, and 4 is by using a modified Tor client as Tor already has all that functionality built into it. In our proof of concept implementation we modified a version of the Orchid Tor client to perform those steps for us.

Also note that it is a requirement that the descriptor download be done based on descriptor-id rather than by `.onion` address as our use of replica numbers is different than that of regular Tor's. This is because in Tor replicas are assumed to all be the same descriptor, just with a different descriptor-id; in our system replicas are like pages of a book, they're likely all different. That being said, each replica (page) *is still* uploaded to, and downloadable from three separate HSDirs.

If you wanted to implement Step 3 yourself, the downloading of the descriptors, you would need to first calculate the HSDirs associated with the descriptor. To do this you would need to get hold of a network consensus document, such as the `cached-microdesc-consensus` document which is downloaded automatically when you run Tor. You would then select the three onion-routers in that consensus who had the `HSDir` flag and whose identity-digest was closest to (and larger than) the descriptor-id of the descriptor. Those three onion-routers are the descriptor's HSDirs. You would then create a circuit ending at one of those HSDirs and make a HTTP GET request of the form:

```
GET /tor/rendezvous2/<descriptor-id> HTTP/1.0
Host: <HSDir-IP>:<HSDir-port>
```

Servers running newer versions of Tor won't accept direct, unencrypted HTTP GET and POST requests for hidden service descriptors. Additionally, it's important to note that if we don't make our requests over a Tor

circuit all anonymity is lost. To achieve this our implementation downloads and publishes descriptors using a modified (Tor) Orchid client. Orchid is a Java implementation of Tor and was used as it was found to be far easier to modify than the main Tor client (which is written in C).

When a bot downloads a hidden service descriptor it must validate it in much the same way that Tor validates descriptors (Step 4). If you wanted to implement this yourself you must first examine the `<public_identity_key>` section of the descriptor to see that it's a valid public key. Next the bot must check that the `<descriptor_signature>` is valid. To do this it decrypts the `<descriptor_signature>` section using the public key in `<public_identity_key>`; it then takes the SHA1 hash of everything proceeding (and including) the line "`signature`", and checks that it matches the decrypted value. If the decryption is successful and the value matches then the descriptor is considered valid.

The following gives an overview of how you could implement the descriptor-publishing functionality:

1. Split the orders into sets of pages, each 9999 or less characters with an added character acting as the "is last page?" marker.
2. Select the first intermediate `.onion` address.
3. For each page calculate a `descriptor-id` using the `.onion` address and the page number as the replica number.
4. Use the `descriptor-id`s and pages to calculate the list of signatures, as shown in Section 3, then append this list of signatures to the last page (or create a new page for them if there isn't enough room).
5. For each page, create a descriptor containing it and its `descriptor-id`, then publish that descriptor to the 3 HSDirs associated with the `descriptor-id`.
6. If more intermediate `.onion` addresses exist select the next one and go back to Step 3. Otherwise end.

In the Step 5 we calculate the descriptor's HSDirs and then publish the descriptor to them. As with the bot client order fetching operation, this function, that is, calculation of the HSDirs and the HTTP POST request are handled by a modified Tor client. If you wanted to do the POST requests manually you would create a Tor circuit to each HSDir in turn, sending a HTTP request over it in the form:

```
POST /tor/rendezvous2/publish HTTP/1.0
Host: <HSDir-IP>:<HSDir-port>
Content-Length: <content-length>

<descriptor>
```

where `<descriptor>` is the descriptor, and `<content-length>` is the length of the descriptor. If a descriptor publish is successful you should receive the message `HTTP/1.0 200 Service descriptor (v2) stored` in return.

It's interesting to note that the server you upload a descriptor to does not actually need to be be the correctly chosen HSDir. Experiments suggest they can be uploaded to arbitrary servers, provided they have the HSDir flag. It helps secrecy, however, if you use the correct HSDirs as the descriptors will appear more legitimate then.

5 Avoiding network congestion

Botnets are huge and have the power to bring a network, such as the Tor network to its knees, even by accident. It is important for the health of the Tor network that the impact of botnets is minimised. If the network is heavily congested then not only do honest Tor users suffer, but the botnet also suffers. To remedy this we suggest that some percentage of bots are selected to become onion-routers, and as such help alleviate the botnet's strain to the network. Here we focus on "breaking even", that is, reducing the botnet's impact to zero. We use the following notation in this section:

h = The number of honest Tor clients in the network. According to the Tor metrics website [10] this value is currently $h \approx 2450000$.
r = The number of honest Tor onion-routers in the network. According to the Tor metrics website [10] this value is currently $r \approx 6000$.
b = The number of bot Tor clients in the network.
p = The number of bot onion-routers in the network.

Our goal here is to calculate $\frac{p}{b}$, the number of bots that we should turn into onion-routers in order to offset the resources used by our botnet.

By far the easiest way to "break even" is to maintain the ratio of Tor clients to Tor onion-routers, $\frac{r}{h}$. Since they only need to do one quick operation we can assume that our bots will use the same amount or less bandwidth and circuit creations as regular clients, so setting $\frac{p}{b} = \frac{r}{h}$ makes it almost certain that the botnet will have a neutral or positive affect on the Tor network. This gives us a ratio of $\frac{p}{b} \approx \frac{6000}{2450000}$, or, for every 1225 bots, 3 should become onion-routers.

With this ratio in mind we're left with the question of how to select which bots become onion-routers. The simplest answer to this is simply by having each flip a biased coin; if heads they become an onion-router, if tails they don't. Due to the vast size of botnets this probabilistic method is sufficient. If 5 million bots each flip a coin biased with the probability p/b we're likely to get a distribution very close to p/b.

As this is the easiest method it's also very approximate. A better estimate would be calculating the number of onion-routers required to offset the number of circuit creations. This method, however, is more difficult as it requires measurements that are not readily available: the average number of circuit creation operations an onion-router participates in during a time period. If we did have such a measurement we could calculate the ratio as: $\frac{p}{b} = \frac{c}{C}$, where C = the average number of circuit creation operations (circuit handshakes) an onion-router takes part in per-second. This value isn't available on Tor metrics so it would require independent measurement to discover. And where c = the average number of circuit creation operations a bot participants in per-second. The value of c isn't as simple as it may seem at first; it requires us to know on average how many times a client fails to download a descriptor before they're successful. As these measurements aren't available we use the previous ratio instead.

6 Defences against this scheme

There are no obvious ways to prevent this scheme from being useful without also harming the Tor network and honest Tor clients. Here we discuss a few potential ways of defending against it and how effective they would be.

6.1 Clear-text hidden service descriptors

Making all hidden service descriptors plain-text rather than allowing encryption would effectively kill our scheme by allowing descriptors to be properly validated. It would, however, also weaken the Tor hidden service protocol. Hidden service descriptors can be encrypted for a reason: in version 3 it's to prevent HSDirs from knowing which hidden service they're serving descriptors for, and in version 2 to provide an optional form of client authentication. Because of these reasons removing encryption is a bad idea.

6.2 .onion blacklisting

You could try to attack this scheme by using `.onion` address blacklisting. An adversary could observe an infected machine and compile a list of all the `.onion` addresses used by the botnet. They could then create a blacklist of `.onion` addresses whose descriptors should be rejected by HSDirs. Assuming they could get all HSDirs to agree to abide by the blacklist this would prevent the scheme from working properly, but with a few modifications this filtering could easily be bypassed.

The reason the scheme can be bypassed is our non-standard use of replica numbers. The blocker doesn't know how many pages are being published, and if non-sequential replica numbers are used they wouldn't know the replica numbers of the pages either. The descriptors could not be blacklisted without knowledge of which replica numbers are used, and hence filtering could not be performed.

Additional difficulties with this scheme arise from the fact that it would require great cooperation from all HSDirs. To achieve such cooperation the blacklisting feature would probably need to be a default part of the Tor software. Whether the Tor project would endorse and implement a hidden service censorship mechanism is up for debate.

6.3 Waiting time between re-uploads

Currently it's possible to upload a hidden service descriptor and then immediately upload a second one that overwrites the first. This lack of a cooldown period allows bot herders to change the current set of orders quickly, without waiting. We could fix this by imposing a limit, such as not allowing a new upload for 30 minutes after the first. Doing that would make the C&C scheme less responsive and hence less lucrative for a potential herder.

The downside is that it also makes hidden services less robust against changes to the network. Take for instance a situation where the introduction points of a hidden service suddenly vanish before the cooldown period is over. This will make the hidden service inaccessible to new clients until the cooldown has finished.

6.4 Strict descriptor format

We could introduce a strict format for lists of introduction points in hidden service descriptors. The most obvious rule we could include is checking that the list of introduction-points is encoded in base64 (not currently checked). Another method could be encrypting each introduction point in the list separately, *i.e.*, three separately encrypted introduction points rather than one large blob. You could then limit the number of introduction points to an exact number (*i.e.*, always three). This would make our scheme more impractical as there would be less space to work with and the space would need to conform to a specific format.

The downside of this is that it reduces the flexibility of hidden service descriptors for regular hidden services as well. Only allowing a specific number of introduction

points removes the possibility of hidden services choosing the number of introduction points they have based on their own specific requirements.

7 Conclusion

This vulnerability is not a significant threat to the Tor network, but does present a new way in which Tor may be misused. Defending against the attack without reducing the strength of the Tor protocol is difficult, but a few measures can be put in place to do so. In all likelihood this vulnerability will continue to exist into the future but will be made less efficient to the point that it is essentially useless.

References

1. arma: [Tor Blog] How to handle millions of new Tor clients. https://blog.torproject.org/blog/how-to-handle-millions-new-tor-clients (2013). Accessed 05 Sept 2013
2. Daswani, N., Stoppelman, M.: The Anatomy of Clickbot.A. In: Proc. of the First Conf. on First Workshop on Hot Top. in Underst. Botnets, HotBots'07, pp. 11–11. USENIX Association, Berkeley, CA, USA (2007). http://dl.acm.org/citation.cfm?id=1323128.1323139
3. Hopper, N.: Protecting Tor from botnet abuse in the long term. Tech. Rep. 2013-11-001, The Tor Project (2013). https://research.torproject.org/techreports/botnet-tr-2013-11-20.pdf
4. Mathewson, N.: Next-Generation Hidden Services in Tor [Draft]. https://gitweb.torproject.org/torspec.git/blob_plain/398c01be40f957c07d23b4ef6192214aee505703://proposals/224-rend-spec-ng.txt (2013). Accessed 23 June 2014
5. msft-mmpc: Mevade and Sefnit: Stealthy click fraud. http://blogs.technet.com/b/mmpc/archive/2013/09/25/mevade-and-sefnit-stealthy-click-fraud.aspx (2013). Accessed 03 Aug 2014
6. msft-mmpc: Tackling the Sefnit botnet Tor hazard. http://blogs.technet.com/b/mmpc/archive/2014/01/09/tackling-the-sefnit-botnet-tor-hazard.aspx (2014). Accessed 03 Aug 2014
7. Nazario, J.: BlackEnergy DDoS Bot Analysis. Arbor (2007)
8. Protect the Graph: Sefnit is Back. https://www.facebook.com/notes/protect-the-graph/sefnit-is-back/1448087102098103 (2014). Accessed 03 Aug 2014
9. Stock, B., Gobel, J., Engelberth, M., Freiling, F.C., Holz, T.: Walowdac-analysis of a peer-to-peer botnet. In: Comput. Netw. Def. (EC2ND), 2009 Eur. Conf. on, pp. 13–20. IEEE (2009)
10. The Tor Project: Tor Metrics. https://metrics.torproject.org/ (2014). Accessed 08 July 2014
11. The Tor Project: Tor Project: Anonymity Online. https://www.torproject.org/ (2014). Accessed 09 July 2014
12. The Tor Project: Tor Rendezvous Specification. https://gitweb.torproject.org/torspec.git/blob_plain/7901fc11a9ecc6e857bf860fecb5ed25bd073378://rend-spec.txt (2014). Accessed 23 June 2014