Security of Ethereum Swarm

Elad Verbin (working version)

Here are some thoughts on the security of Swarm, and how to ensure it. Lots of input and knowledge from the Swarm team is contained here. All mistakes are mine. Thanks to Viktor, Aron, Dani, and all the rest of the Swarm team!
Please comment directly in this document, or contact me at elad.verbin@gmail.com, with any feedback, questions, etc. . This is just a first draft with some ideas, and I intend to cooperate with you guys and keep working on these directions.

Section 1. Random Block Drops, and Swarm Resilience Calculation

1.1. Resilience Calculation.

Suppose swarm opts to use a CRS code with n=128, m=112, k=16 (current parameters in the orange paper (here, Section 3.2.1)). i.e. each 112 4K-byte chunks (a "blob") are turned into 128 blocks (AKA "pieces") of 4K bytes each. These blocks are spread for storage throughout the network.

Suppose Swarm suffers the simplest model of loss: randomly-distributed block drops. Say a fraction p of all blocks are dropped.
Question: What is the chance that a single blob is lost?
Answer: For a blob to be lost, k+1 of its blocks need to be lost. The chance for this to happen is equal to P_p = Pr[BinomialDistrib(n, p) >= k + 1]. Substituting n=128, k=16, and putting various values of p, this gives:
1. with p=0.05 this gives P_{0.05} = 2.3 * 10^{-4}
2. with p=0.02 this gives P_{0.02} = 10^{-9}

Attack Threshold 1:
Consider a 4GB file stored on swarm. Such a file consists of ~10K blobs. Under 4.5% drop-rate, each of these 10K blobs has a chance of 2.3 * 10^{-4} to be lost. Overall, such a file will be lost with probability 90%.[1] That means with p=0.05, we are likely to lose almost all files of size >= 4 GB.

Thus, even a "5% attack", i.e. an attack where a random 5% of all blocks die at once, will cause a catastrophic event on Swarm where almost all large files become unrecoverable.

[1] 1- (1 - 2.3 * 10^{-4})^{10^4} = 0.9

Attack Threshold 2:
Consider a “typical” case where Swarm maintains, in total, 4000TB of data, ie ~10^{10} blobs. Then a p=2% drop-rate/attack will cause each blob to be unrecoverable with probability 10^{-9}. Looking over all blobs, we get that almost certainly, some blob out there will become unrecoverable, so some file on the network will die.

1.2. Audit-and-fix methodology: ensuring resistance to block drops.

We conclude that even at the most favorable attack model (or simply “natural disappearance of nodes” model), where random pieces disappear and no piece drops are “targeted”, all files need to be audited before 2% drop-rate is reached.

How should audits be timed? The conceptually simplest way to think of the timing of audits is to imagine a separate “drop timer” being placed on each file at its creation date. Each time a block, anywhere in the system is dropped, the timer gets increased by (1.0 / current_number_of_blocks_in_the_system). When the timer reaches 0.02, an audit on this file is triggered, and all blocks of this file are checked. Performing an audit resets the drop timer to zero.

Audits performed this way should generally prevent file drops, unless:

1. A “correlated drop event” happens. E.g. if AWS S3 shuts down surprisingly, and many nodes store blocks on AWS, then the 2% (or even 5%) drop can happen all at once, and audits can’t do anything about it.
   (This can also happen if AWS sees Swarm as competition, and decides overnight to claim that “Swarm helps terrorists” and shuts down access to all Swarm node data stored on AWS.)
2. A deliberate attack: if e.g. a state attacker wants to damage the reputation of Swarm, it can create more and more nodes, until its nodes are responsible for 5% of all blocks. Then it can take all nodes offline, and damage the entire Swarm payload.

Also note that a Swarm “5% attack” is actually way more destructive than a “51% attack” on Ethereum. That is because in Swarm, once the files are deleted, they’re gone, and there’s no turning back. While in Ethereum, if a “51% attack” was launched, the damage can be undone by agreement of the community: the state can be rolled back.

A “5% attack” can also be used as a ransom attack: I perform a 5% attack, then ask for lots of money to give the files back. Nodes can even coordinate via public channels to perform such an attack: they join a smart contract, that asks for ransom and spreads the ransom among all nodes that went offline at “D-day”. If not enough stake/deposit is taken, then the rational action is to join such a contract.

Note, also, that it's not needed for attackers to make themselves known ahead of time. Attackers may hold all the blocks diligently and perform their duties until, one day, when they have amassed 5% of all blocks, they take the entire system down.

Section 2. Attackers that are able to choose their Kademlia addresses

2.1. Which attack model to use?

The above math considers just the case where block drops are selected randomly. But Swarm is a complicated cryptoeconomic system, and there might be other vulnerabilities. For example: what if attackers get to choose their own Kademlia addresses? In that case, attackers can try to target specific files.

To analyze this, we need to define an “security model”. This consists of:

- What are the resources the attacker has?
    - In terms of money, time, processing power, network capacity, cooperation, maybe even political power that allows it to influence honest nodes
    - What “savvy” does the attacker have? E.g. do we let it infinitely snoop on all unencrypted internet traffic?
- What “bad events” are we trying to avert?
    - Are we trying to avert any file being killed? Or only high-priority files (which are explicitly marked, and for which the “storage charges” are higher)?
    - Are we trying to ensure high uptime, or just eventual availability?
    - This is really a “business question”. It asks stuff like “are we willing to explain to our users that we chose to make the service 10x cheaper, at the expense of being vulnerable to a consortium of powerful state actors.

2.2. Description of a possible attack.

From now on, I’ll analyze a model where attackers have access to all unencrypted traffic, and to anything in the protocol that is not explicitly securely hidden. So they know:

- which nodes hold which blocks
- which blocks belong to which file
- which addresses are currently occupied by a node
- etc. .

Under this assumption, **attackers can pretty much choose their Kademlia address at will.** They create nodes again and again, until they get the right address (or at least an address close enough to where they need it). Generally speaking, if there are N nodes in the entire system,

then creating O(N) addresses would probably be enough for any practical need. Since creating an address is as easy as creating a public-private key pair, this cost is not prohibitive.

This means that an attacker can target a file just by choosing an arbitrary blob of that file, then choosing addresses that are closest to the blocks of that blob. The “implicit redundancy” of Swarm cannot be relied on to save the file, since we assumed that the attacker knows which nodes hold which blocks, and it can just engineer the addresses such that sooner or later, all copies of those blocks are under the attacker’s control. (Also, we should not reply on implicit redundancy to save us from attacks, because it is not designed for that purpose.)

Thus, we see that files can be cheaply attacked in the current design of Swarm, by a wily attacker.

2.3. How to defending from targeted attacks.

We see that if node addresses can be cheaply chosen by an attacker, files can be cheaply attacked. There are several ways to try to avert such attacks:

- Making address-selection more costly (in time or money)
    - At an extreme, attack selection is so costly that it will realistically only be done once, or few times, per node, and this puts us back in the easier model that we dealt with in Section 1
- Hiding information from an attacker
    - If we manage to reliably hide from the attacker the redundancy information, or the network topology, or even the identity of which blocks belong to each file, then they might not be able to perform such attacks.
    - This will likely require some cryptographic protocol design.
    - The problem with this approach is that “hiding stuff” is a difficult way to convincingly ensure security. The arguments for “positive security” or security proofs will have to rely on arguments like “the attacker does not know X”, and these statements are hard to reason about. (And, indeed, are hard to ensure in the real world. Information leaks!)

Here I will concentrate on the approach of making address-selection more costly. There are many approaches to this. Here are two approaches:

- **Make the node deposit non-refundable.** We take a deposit for registering a node, and only then we select the node address. This address is generated by a PRG whose seed depends on the most recent Ethereum-chain nonce. This makes sure the address is not predictable in advance. So the node first pays a deposit, then gets assigned a random address, and it never gets the deposit back.
- **Make the node go through an “internship period”.** When a new node joins, we ask them to put a (refundable) deposit. We then ask them to serve the network’s needs for a week as a real node would, but we don’t trust them to be custodians of blocks: any block held by them, will also be held by an older, trusted, node. After this week, the node gets

"certified", and needs to choose a new, random address (by a PRG seeded by the Ethereum nonce). This will be the address of this node from now on, and the node will be trusted to be a custodian of blocks.

With these approaches, we see that a real spending of resources in time and/or money needs to be made in order to gain an address. Other ways of "extracting a toll" for selecting of "certified addresses" are also possible. I'll be happy to help sift through them and decide which approaches are most promising. (Aron had some ideas.)

The problem with these approaches is that it still allows an attacker to gain addresses by "buying" them in a secondary market: If getting a node started is costly, then people will want to sell their nodes. If many people want to sell their nodes, the attacker can just buy the nodes with the addresses they need. I don't know how to deal with this problem

2.4. Summary of Section 2.

No summary yet. Lots of work to be done here.

Part 3. Other kinds of attacks

In the last sections we only considered the kind of attacks that tries to eliminate files by removing some of their blobs. But Swarm is a complex protocol, and there might be many other ways to disrupt it. Here are just a few :

- Some nodes can place themselves in strategic locations in the network, and return wrong results (e.g. "your file does not exist")
- DoS-type slowdowns
- During the Kademlia routing process that finds the node that is closest to some address, the queried nodes on the way can give the wrong answers.

Those are just some forms of attacks that quick come to mind. A more thorough process should map these comprehensively.

Appendix: Further Thoughts and Some Musings by Elad

(unstructured and partially basic or dumb; feel free to ignore)

Follow-up to Part 1.

From part 1 we see that audits are a decent tool, except in “coordinated-loss” scenarios. What tools do we have to combat such situations? Some obvious tools that come to mind are:

- Obfuscated blocks
    - Make it hard to see which file each block belongs to. Maybe by using cryptography. This will make it harder to attack one file in particular. But it doesn’t make it more difficult to cause a “catastrophic event”
- Unpredictable redundancy
    - Make the amount of redundancy of each file unpredictable. Ditto: don’t stop “catastrophic event”
- Losing “resources” when block is dropped. “Resources” might mean a stake/deposit, or time and effort.
    - This indeed gives a way to stop a catastrophic event.

From all the ways I thought of, the best way to combat catastrophic events are by taking a deposit. The deposit put up by a node needs to be proportional to the amount of blocks stored in the node. (Or, for simplicity, just make a node always store at most 4GB. Want to store more? Open another node on the same computer.)

Let’s assume each node stores exactly 4GB. How much should the deposit be? Well, to cause a catastrophic event, 5% of nodes have to disappear at once. So The deposit, if taken in money, should reflect that: if we want the cost of causing a catastrophe to Swarm to be $1B, then the total deposits put by all Swarm nodes should be $20B. Is this too much? I’ll let you judge.

This ratio of 20-to-1 between the deposits put up by the entire network to the deposits that much be put up by the attackers, is directly derived from the fact we’re talking about a “5% attack”. How can we make this 20-to-1 ratio more favorable? We have two options:

1. Add redundancy. So maybe we want to change the parameters of the CRS code to be n=128, m=64, k=64. This will necessitate something like a “30% attack”, so a 2-to-1 ratio. Much better! The cost will be that we’ll be working with a 100% redundancy rather than 15%.
   Personally, I think this is a good move.
2. Increase the block size of the code. So maybe we change the parameters of the code to be n=1280, m=1120, k=160. This will probably bring us close to needing a “10% attack” for a catastrophic event. It keeps the same redundancy, but has some nuanced drawbacks.
   (Aron will know more. I think there’s a claim that this dramatically hurts computational efficiency but I’m not sure)