A Deep Dive of HOW Profanity Caused Wintermute to Lose $160M

2022-09-24

By Max

Background

On September 20, Evgeny Gaevoy, CEO of liquidity provider Wintermute, announced a $162.2 million loss due to an exploit. The "private key" of an address is thought to have been brute-forced, causing an uproar within the community. According to The Block, on September 19, hackers also utilized the same vulnerability to steal 3.3 million from several Ethereum addresses that were generated using Profanity.

Our security team followed up with this incident and discovered that the Wintermute affected address was generated using Profanity and vanity addresses starting with 0x000000 can be brute-forced within a reasonable amount of time. 1inch has previously published a document disclosing the potential security threat of this vulnerability. So, how exactly did these attacks happen? What role does Profanity play? In this article, we will be providing a full analysis of these attacks from the following aspects.

- What Is A Vanity Address?
- How Does Profanity Generate A Vanity Address?
- How Ideally Secure is Profanity?

- Risks Associated With Profanity
- Intro Into Profanity Algorithm
- Principles Of Private Key Attack on Vanity Address
- Examples Of Private Key Attack On Vanity Address
- Attack Efficiency Analysis
- Security Improvement Proposals
- Afterthoughts

What is a vanity address?

Vanity addresses are addresses with predefined, personalized patterns, such as addresses with a certain set of numbers in the beginning.

- Beginning with eight 8's:

```
0x88888888bc27358faea388cdf91fa9b676068207
```

- Beginning with seven 0's:

```
0x0000000925e311792debae85befaa946200ffc67
```

- Beginning with a specific word:

```
0xdead9b096ec34c35e45b5a2aab5337805916ac1e
```

- Docker image:

```
0x5b4d6554bd4d 89dfcd0bb0dcfd98 c3e73ab05f69
```

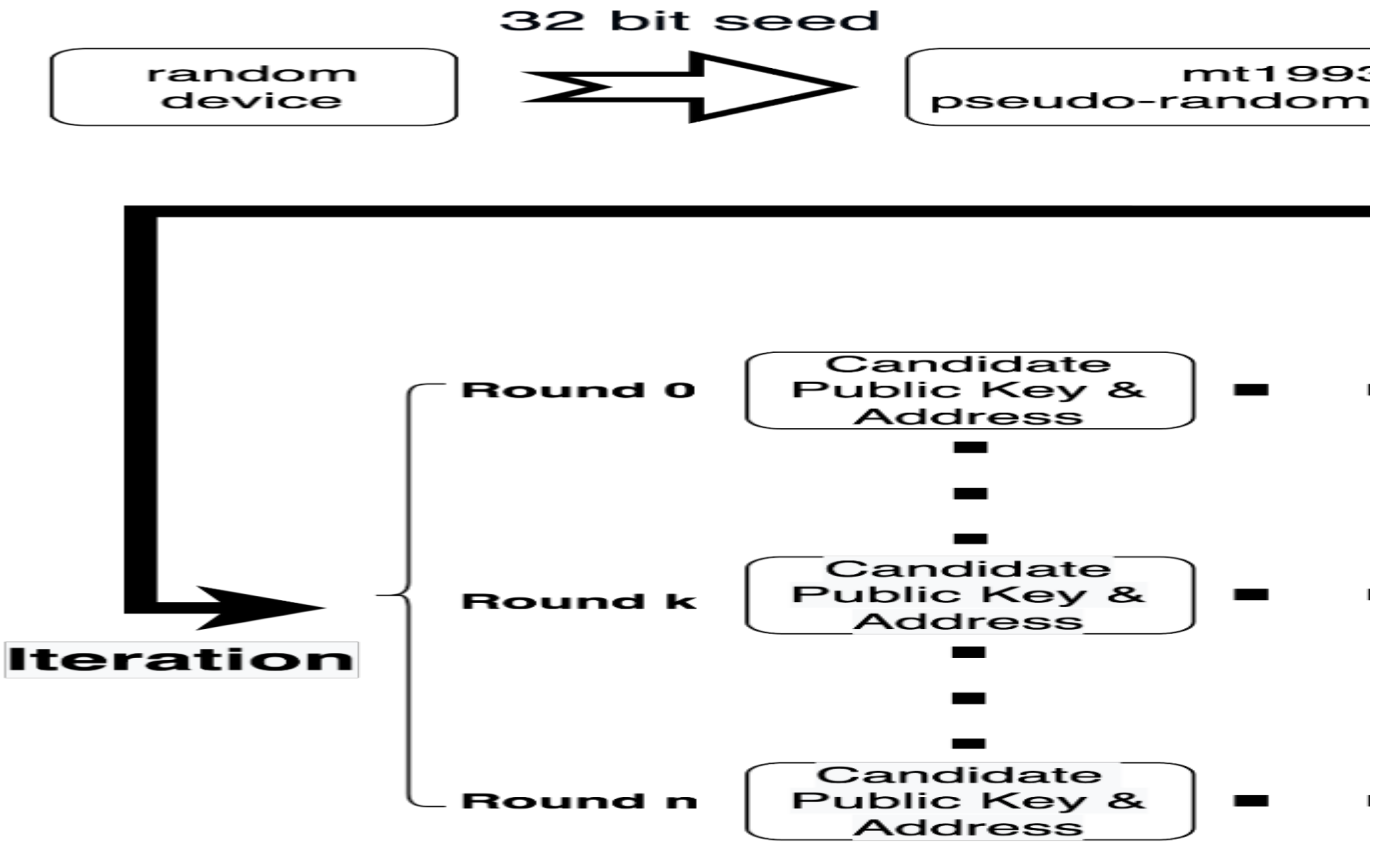
These vanity addresses have characteristics that distinguish them from other randomly generated Ethereum addresses.

Developers build a variety of open source tools to generate these vanity addresses, with Profanity being one of the most popular for Ethereum addresses. So, how does Profanity generate a vanity address?

How Profanity generates a vanity address?

How Profanity generates a vanity address?

Let's take a closer look at how Profanity generates a vanity address. Below is the flow we concluded from the Profanity Repository.



The generation steps are as follows:

1. Retrieve a secure random number (from hardware or an entropy pool) as the seed for the pseudo-random generator mt19937.
2. Use pseudo-random generator mt19937 to generate the seed private keys and then calculate the seed public key.
3. Call the OpenCL parallel computing platform, and then call the multi-iteration algorithm on the seed public key to calculate a large number of candidate public keys and Ethereum addresses.
4. Call the vanity address filter, screen out the vanity addresses, and output the vanity private key and vanity address.

As one of the most popular vanity address generators for Ethereum users, Profanity supports parallel computing where users can have a variety of vanity addresses and enjoy incredibly efficient key generation. However, this ease of use paved the way for subsequent attacks.

Ideally, how secure is Profanity?

Ideally, how secure Profanity can be? Let's first take a look at how long it will take to brute-force a vanity address's private key.

Vanity Address:

```
0x88888888bc27358faea388cdf91fa9b676068207
```

We can approximate the overall number of computations needed. Since the original seed range is $(0, 2^{32})$, starting with 0x88888888 and assuming the user chooses to use the first matched address, then the number of computations required is $2^{32} * 2^{32} = 2^{64}$, demanding a significant amount of computing power.

Estimated time for brute-force: Using the configuration on my personal device as a test, the average time to compute a new vanity address takes around 10s (estimation of 3 times), then the time needed (in months) to brute-force attack the private key of a vanity address is $2^{32} * 10/(3600 * 24 * 30) = 16570$.

If I combine the computational power of 1,000 PCs with the same specifications as mine, it will still take around 16 months of working nonstop to successfully brute force attack the private key.

This seems like a very secure method, but is it really?

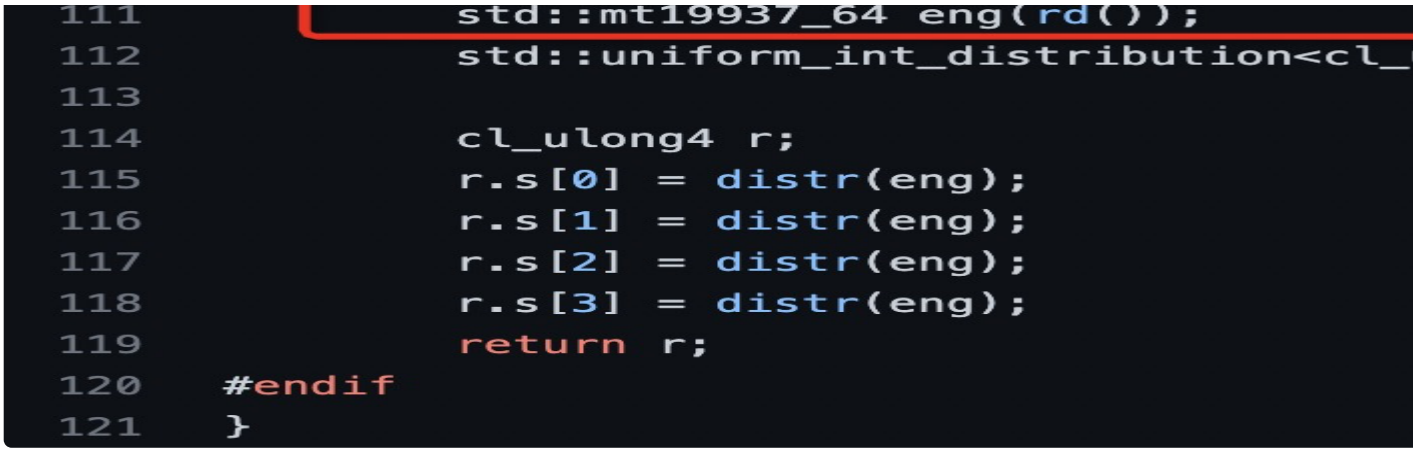
Risks associated with Profanity

At first glance, there appears to be no apparent issue with calling secure random numbers and performing a lot of calculations. However, the hacker was still able to successfully carry out attacks, so how was this possible?

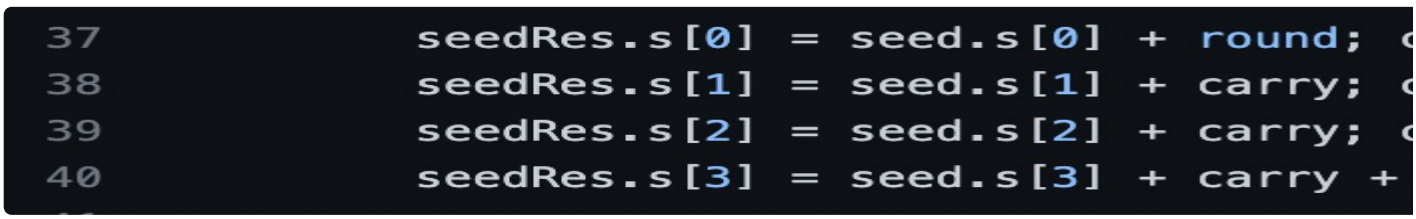
There are at least 2 major vulnerabilities in Profanity:

- When retrieving random numbers from hardware or an entropy pool, only 32 bits are retrieved, which is quite small. The common private key has a length of 256 bits, which is significantly greater than 32 bits.

```
100   cl_ulong4 Dispatcher::Device::createSeed(
101   #ifdef PROFANITY_DEBUG
102           cl_ulong4 r;
103           r.s[0] = 1;
104           r.s[1] = 1;
105           r.s[2] = 1;
106           r.s[3] = 1;
107           return r;
108   #else
109           // Randomize private keys
110           std::random_device rd;
```



- During parallel multi-round calculation, the iteration algorithm does not use one-way functions (such as hash functions). One-way functions are more secure. Not using them can lead to the possibility of reverse traceability.



While a single vulnerability might not be fatal, the combination of the two can lead to serious consequences.

Introduction into Profanity algorithm

Before diving into how the attack happened, let's take a closer look at how Profanity works.

Step 1: Compute Seed Public Key

From random device to seed public key, there are two key steps:

- Retrieve a 32-bit secure random number (from hardware or an entropy pool) as the seed for the pseudo-random generator mt19937.
- Use the pseudo-random generator mt19937 to generate seed private keys and then calculate the seed public keys.

For the 32-bit secure random number, we can define the random seed as $R = [0, 2^{32})$; Since mt19937 is essentially a deterministic algorithm, the calculation of the seed private key is deterministic, thus, the seed public key is also deterministic.

As shown below:

$$R = [0, 2^{32}) \ \ SeedPriv = mt19937(i), i \in R \ \ SeedPub = SeedPriv * G$$

To simplify the formula, we define the function as below: $f(x) = mt19937(x) * G$

Record the seed public key as $Set_{SeedPub}$, then

$$Set_{SeedPub} = \{f(i) \mid i \in R, R = [0, 2^{32})\}$$

Of course, the number of seed public keys is 2^{32}.

Step 2: Calculate Candidate Public Key

Profanity begins with the seed public key and executes numerous rounds of iteration operations concurrently to generate a series of candidate public keys. The entire procedure is simply a public key search and traversal algorithm that revolves around the specified seed public key. The iteration is based on a two-dimensional linear search operation using the round number ($round$, as r) and the in-wheel sequence number ($foundID$, as fid).

The iterative algorithm is as follows: $CandidatePub_{r,fid} = SeedPub + \Delta(r, fid) * G$

Iterative offset function is defined as: $\Delta(r, fid) = r * k_1 + fid * k_2$ $k1$ and $k2$ are two constants, G is the base point on the elliptic curve.

As a result of the iterative calculation, the set of candidate public keys can be defined as:

$$CandidateSet_{Pub} = \{CandidatePub_{r,fid} \mid r \in [0, r_{max}), fid \in [0, fid_{max})\}$$

Substitute $CandidatePub_{r,fid}$ into the definition, i.e.:

$Set_{CandidatePub} = \{SeedPub + \Delta(r, fid)G \mid r \in [0, r_{max}), fid \in [0, fid_{max})\}$ where r_{max} and fid_{max} each represents the upper bound of round number and lower bound of in-wheel sequence number that helps us specify the size of the iterative search space.

$SeedPub$ taking different values will generate different sets of candidate public keys.

I.e.: $SeedPub = f(2)$, then:

$$Set_{CandidatePub} = \{f(2) + \Delta(r, fid)G \mid r \in [0, r_{max}), fid \in [0, fid_{max})\}$$

Note: The following equation is valid based the definition of the iterative offset function:

$$\Delta_{r_1,fid_1} - \Delta_{r_2,fid_2} = \Delta_{r1-r2,fid_1-fid_2}$$

Step 3: Filter Vanity Address

ETH addresses are calculated based on each candidate public key. Then we can use pattern

filters to sift out vanity addresses. Here, we denote a set of vanity addresses as $Set_{VanityAddr}$.
So far, the vanity address generation algorithm of Profanity can be concluded as below:

Principles Of Private Key Attack on Vanity Address

Next, we'll look at how to attack the private key mechanism in the figure above.

Step 1: Exhaustive seed public key set

The seed public key is set to $Set_{SeedPub}$ and upper limit of the number of elements is likewise set to 2^{32}. To summarize, the full set of $Set_{SeedPub}$, including its corresponding private key, takes up around 300 G of storage space, which can be easily stored on a single machine.

Step 2: Calculate the real vanity public key

Find the vanity address and recover the vanity public key from the signature in any online transaction. Set it as $VanityPub_0$. See public key recovery for specific algorithms.

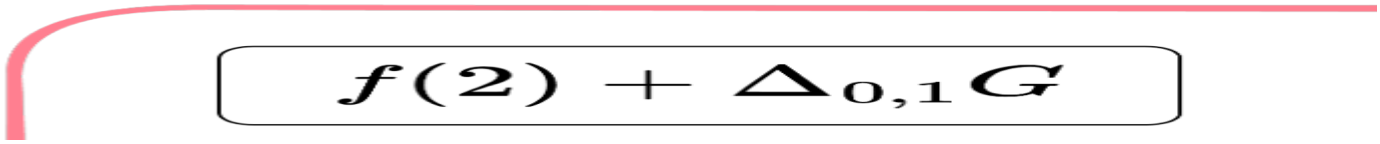
Step 3: Calculate a shift set of the candidate public key set

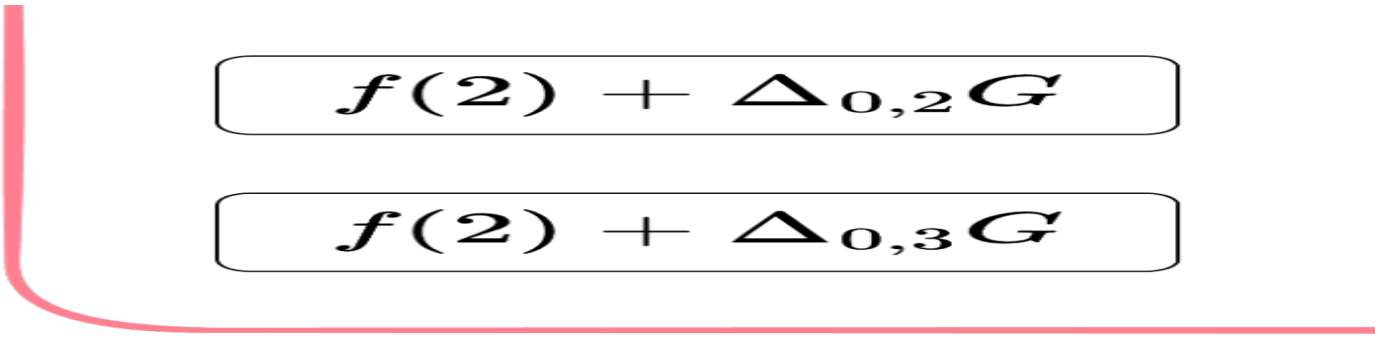
We defined the shift set of the candidate public key set with respect to the vanity public key $VanityPub_0$:

$$ShiftSet_{VanityPub_0} = \{VanityPub_0 - \Delta_{r,fid}G \mid r \in [0, r_{max}), fid \in [0, fid_{max})\}$$

For better illustration, let's assume there are 3 rounds and the number of iterations is also 3.

The following is a the candidate public key set based on the seed public key $f(2)$:

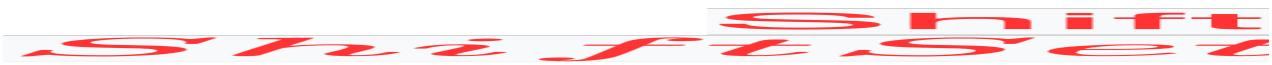


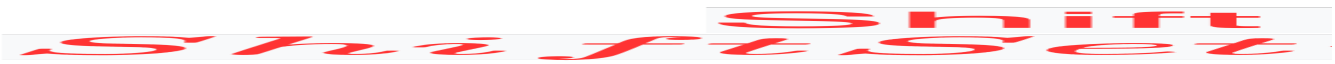


A certain public key in the candidate public key set will be screened out as a vanity public key. Let's set the selected vanity public key as:

$$VanityPub_0 = f(2) + \Delta_{1,2}G$$

The following will demonstrate how to compute $f(2)$ from the shift set $ShiftSet_{VanityPub_0}$.



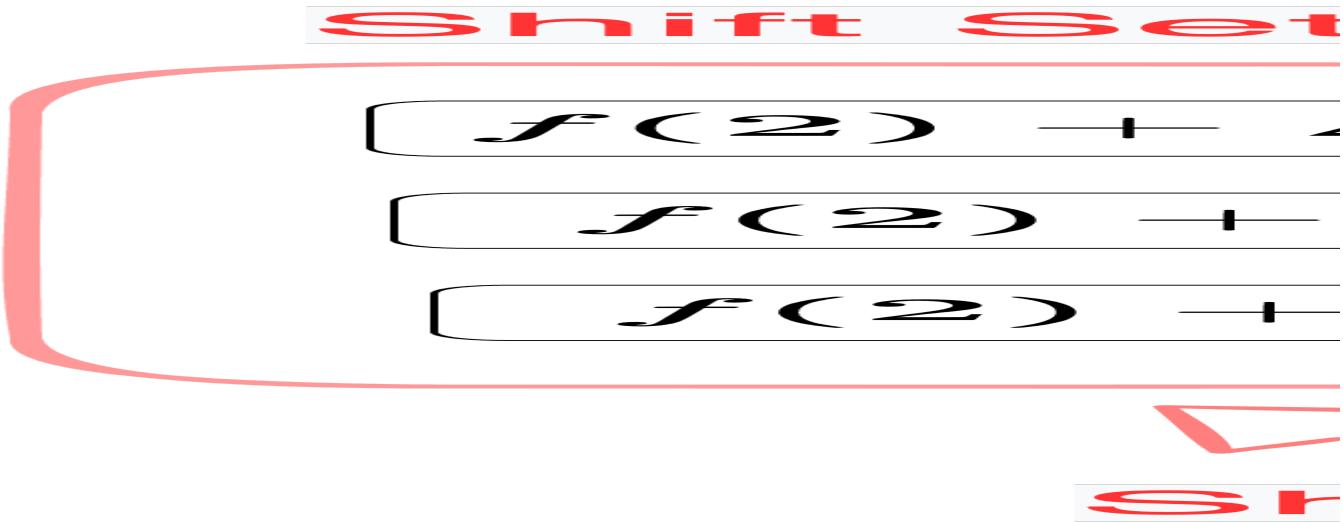


$$\{f(2)+\Delta_{-1,-1}G, f(2)+\Delta_{0,-1}G, f(2)+\Delta_{1,-1}G, f(2)+\Delta_{-1,0}G,$$
$$f(2), f(2)+\Delta_{1,0}G, f(2)+\Delta_{-1,1}G, f(2)+\Delta_{0,1}G, f(2)+\Delta_{1,1}G\}$$

It contains the seed public key $f(2)$ corresponding to the vanity public key. A careful person can find that the offset of the vanity private key compared to the seed private key is $-\Delta_{1,2}$.

Tips:

- Why is it named "Shift Set"? This is because the shift set and the candidate public key set are two matrices in a two-dimensional space. The two matrices are of equal size, may be adjacent, or some elements may overlap. It appears that the shift set can be computed by shifting the candidate public key set based on the vanity public key. As shown below.



- Why compute a shift set? Because the shift set includes the seed public key. As illustrated in the picture above, $f(2)$ is the seed public key.

Step 4: Calculate intersection set

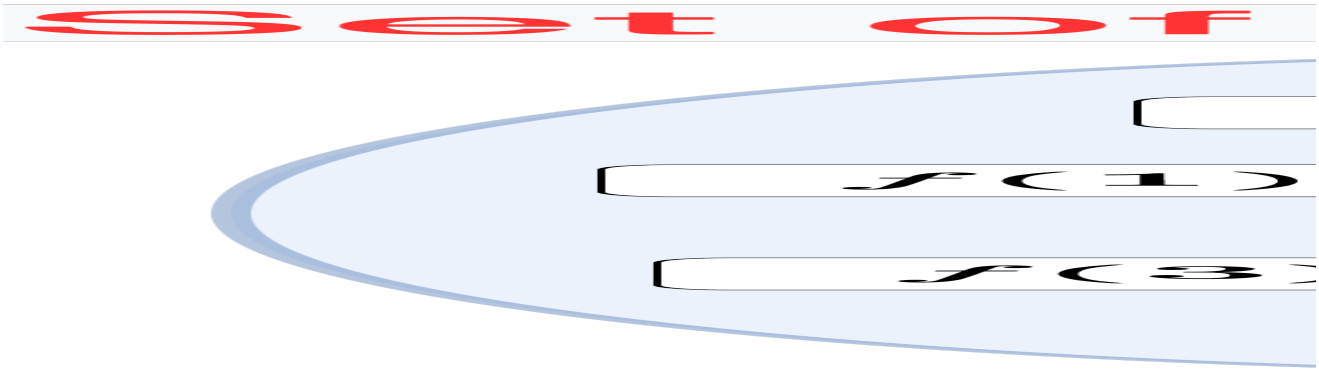
Denote the iterative search range for the computation of the shift set as r_{max}, fid_{max}, the original iteration coordinates of the user's vanity public key as r, fid, then the following

Proposition：If $r_{max} > r$, $fid_{max} > fid$, then there must be an intersection between the shift set calculated based on the vanity public key and the seed public key set.

This notion assures the discovery of the seed public key associated with the vanity public key.

Calculate the intersection $InterSect = Set_{SeedPub} \cap ShiftSet_{VanityPub}$.

There exists at least one element in the intersection, which is a seed public key, and it can be defined as $SeedPub_0$, and the corresponding private key can be obtained from the pre-stored data in **Step 1**. Since $SeedPub_0 \in ShiftSet_{VanityPub}$, so, private key offset Δ of $SeedPub_0$ in comparsion to $VanityPub$ is also recorded in **Step 3** and can be obtained by querying the data.



Using **Step 3** as an example, as shown in the figure above, the intersection set is $InterSect = f(0)$, the query shows the private key offset of the seed public key compared to the vanity public key as $\Delta = -\Delta_{1,2}$.

Note: There are numerous implementation methods for computing intersection sets. For example, if the amount of data is minuscule, you can directly check the database. If the amount of data is enormous, you can solve it by combining some sophisticated algorithms.

Step 5: Calculate the vanity private key

At this point, the private key of the vanity public key $VanityPub$ can be easily calculated:

$$VanityPriv = SeedPriv - \Delta$$

Using examples from **Step 3** and **Step 4**, the vanity private key can be directly calculated:

$$VanityPriv = Priv_{f(0)} - (-\Delta_{1,2}) = Priv_{f(0)} + \Delta_{1,2}$$

Examples Of Private Key Attack On Vanity Address

Now we use a specific example to show how to obtain the private key of a vanity address. The

vanity address we are attacking contains the word “dead” within it.

```
0xdead9b096ec34c35e45b5a2aab5337805916ac1e
```

Step 1: Exhaustive seed public key set

Pre-generated seed public key set $Set_{SeedPub}$ and store it. Note that the private keys corresponding to all the seed public keys are also stored at this time.

Step 2: Calculate the real vanity public key

Find the on-chain transaction for that address, recover the public key from the signature, as shown below:

```
0x0488dff9528cc2fc582e11688abce90cd84d8c805424fa3c761f50ad96b877a8cf3c3b0796ec09
```

Step 3: Calculate the shift set of the candidate public key set

Record $VanityPub_0$ as the public key obtained in **Step 2**, specify $r_{max} = 4, fid_{max} = 0x10000$.

Now the number of shift set elements is: $r_{max} * fid_{max} = 262144$, which is a rather small set and signifies a limited search range.

Calculate the shift set: $ShiftSet_{VanityPub_0} = \{VanityPub_0 - \Delta_{r,fid}G\}$

(As $r \in [0, 4), fid \in [0, 0x10000))$

Tips:

- In the process of calculating the shift set, the offsets of all elements in the shift set (that is, the public key point) compared to the vanity public key are also recorded.
- Regarding the selection of r_{max}, fid_{max} in shift sets, the size of the shift set can be determined based on the difficulty of the "vanity address."

Step 4: Calculate intersection

Calculate the intersection of $Set_{SeedPub}$ and $ShiftSet_{VanityPub_0}$ to get the set $\{SeedPub0\}$.

The seed public key $SeedPub0$ in the set is:

```
0x04f316acd6890440bb7ed841e9c9d0a69dbd3545ef390947ad55248cd90719ff84e897f3359caf
```

By querying the pre-stored information, the corresponding seed private key $SeedPriv$ is:

```
0x045c2f14d04b94b99f64c1c36e984311d2fdb49c9b39f8aa272b92da72e323e0
```

Since $SeedPub0 \in ShiftSet_{VanityPub0}$, the private key offset of $SeedPub0$ compared to $VanityPub$ is also recorded in **Step 3**, giving us the result below:

$$\Delta = -131222811778258561367987177892156266428619740566520641492090882$$

Step 5: Calculate the vanity private key

$$\begin{aligned} VanityPriv &= SeedPriv - \Delta \\ &= 0x045c2f14d04b94b99f64c1c36e984311d2fdb49c9b39f8aa272b92da7 \\ &= 0x045c2f14d04be6629f64c1c36e984311d2fdb49c9b39f8aa272b92da7 \end{aligned}$$

Calculate the vanity public key from the vanity private key and verify if it's correct. If it's correct, then the attack was successful.

Attack Efficiency Analysis

Detailed analysis

There are 5 steps in total, basically **Step 2** and **Step 5** are not time-consuming. Since **Step 1** can be predetermined, time can also be reduced from **Step 1**. In the end, the main time-consuming steps are **Step 3 and Step 4**.

In order to assess the scale of this operation, we can define the upper limit of the number of rounds for the candidate public key set as r_{max}, the upper limit of the order within a round as fid_{max}.

Step 4 requires us to solve the intersection problem between two sets, 2^{32} and $r_{max} * fid_{max}$. Compute intersection is a sophisticated problem with complex solutions, therefore we will not be elaborating further here.

Let's focus on **Step 3**. **Step 3** is to calculate a shift for the candicate set. Ideally, the scale is the

same as the candidate public key set, so the number of calculations is $r_{max} * fid_{max}$. What exactly does this mean? **If we ignore the time it takes to solve the intersection set, the time it takes to attack a private key is nearly the same as the time it takes to produce a vanity address.**

Tips:

- The shift set's design and computation have a direct impact on the attack's efficiency.
- Because the candidate public key set is unknown in the actual attack, we can estimate the size of the candidate public key set based on the vanity difficulty in order to design the shift set.
- The cardinality of the shift set may be slightly larger than that of the candidate public key set.

Case analysis

Here's a thought: After some experiments, it is found that most of the time, for the first 2 to 3 vanity addresses under a specified mode in Profanity, the round number is less than 2^4, the in-wheel sequence number is less than 2^{22}.

This indicates the cardinality (total number of elements) of most of the shift sets is $|ShiftSet| <= r_{max} * fid_{max} = 2^4 * 2^{22} = 2^{26} \approx 6kw$. This means that, in most cases, 60 million is the maximum attempt required to successfully crack the private key.

There are also many cases where the shift set is even in the order of millions making it easier to decipher.

Overall efficiency

The primary finding is that these attack efficacy rates are quite high, and practically all Profanity vanity address private keys can be decrypted without paying a significant price.

- If we ignore the time it takes to solve an intersection set, it would take the amount of time to decrypt the private key closing to it does to generate a vanity address. This is obviously terrifying and dangerous for users of the profanity tool.
- The probability of private key decryption is greatly increased by the user's poor security practices. In general, the earlier the Profanity-generated address, the smaller the search space, the greater the risk, and the easier it is to crack. Since most users prefer to use the first generated vanity addresses, the private keys of most Profanity users' vanity addresses can be decrypted at any time.

Security Improvement Proposals for Profanity

There are two main suggestions:

- When a random number is obtained from the hardware or the entropy pool as a seed, 256 bits are directly taken out.
- When performing multiple rounds of iterative computations in parallel, use a one-way function (such as a hash function).

Afterthoughts

If one-way functions are used in Profanity to generate candidate public keys, is it still safe?

The answer is NO. One-way function used to generate the alternate public key does prevent the attack method proposed in this paper, but it does not prevent the birthday attack, since the seed length is 32 bits, then the probability of collision is 50%,
$n(0,5,H) = 1.1774 * \sqrt{(H)} = 1.1774 * \sqrt{(2^{32})} = 77162.$

This demonstrates that if you apply the same vanity address technique, as long as the number of Profanity users is close to 80,000, there is a 50% chance of creating duplicate private keys.

If the length of the random seed is increased to 64 bits, will it be safe then?

The answer is still NO. 64 bits are not a long enough random number, and there is also the problem of birthday attacks, while the number of people with a 50% collision probability has increased to about 5 billion, it is still not safe enough.

Can we still use vanity addresses?

First, vanity addresses generated by Profanity have fatal risks, so we strongly recommend that all Profanity users transfer their assets to other addresses.

Second, we can still use vanity addresses. Not all vanity addresses have these issues.If you choose security over convenience and generate with reliable algorithms, you can create secure vanity addresses.

Any advice on secure generation of private keys?

The private key is generated at random. To make the private key sufficiently secure, it must be sufficiently random. There are numerous techniques to obtain a random, secure private key.

- If a certified secure hardware is available, secure random numbers generated by the secure hardware can be used.
- For a sufficiently complex computing platform, including common types of PCs and mobile

terminals, the environmental noise is complex enough, and the entropy pool and pseudo-random number generator built into the platform are sufficiently random.

- If it is a simple platform, such as some hardware wallets, you need to import the random seed from outside. The methods include but are not limited to importing random files, photos, voices, videos, and so on.
- There is also a safer way to aggregate randomness across multiple platforms via MPC. Generate private key shards from multiple platforms based on MPC. Even if the private key of a single platform is not random enough or is hacked, the private key of the entire MPC wallet is still safe.

Previous
Comparative Analysis on 5 Major Digital Asset Custody Solutions

Next
Analysis On Ed25519 Use Risks: Your Wallet Private Key Can Be Stolen