Association for Information Systems
AIS Electronic Library (AISeL)

2015

How Perfect Offline Wallets Can Still Leak Bitcoin Private Keys

Stephan Verbücheln

Humboldt-Universität,Germany, verbuech@informatik.hu-berlin.de

Follow this and additional works at: http://aisel.aisnet.org/mcis2015

HOW PERFECT OFFLINE WALLETS CAN STILL LEAK BITCOIN PRIVATE KEYS

Stephan Verbücheln, Humboldt-Universität, Berlin, Germany, verbuech@informatik.hu-berlin.de

Abstract

ECDSA has become a popular choice as lightweight alternative to RSA and classic DL based signature algorithms in recent years. As standardized, the signature produced by ECDSA for a pair of a message and a key is not deterministic. This work shows how this non-deterministic choice can be exploited by an attacker to leak private information through the signature without any side channels, an attack first discovered by Young and Yung for classic DL-based cryptosystems in 1997, and how this attack affects the application of ECDSA in the Bitcoin protocol.

Keywords: ECDSA, Bitcoin, Digital Signatures, Kleptography.

1 Introduction

Bitcoin is decentralized payment scheme first described in a publication by pseudonymous author Satoshi Nakamoto in (2008). Because its design is based on cryptographic protocols, the term *cryptocurrency* has become common to refer to systems like Bitcoin and its variants. In a nutshell, users hold public-private key pairs, where the public keys function as an account numbers, and the private keys enable them to spend money from those accounts.

Bitcoin has become very popular and commercially adopted in recent years. The exchange rate has risen to a point that the value of all coins in circulation adds up to \$5 billion as of November 2014 (peaking even higher before). Because of the value of Bitcoin assets, it has become more and more important for users – especially large-scale users like online shops – to protect their private keys, as stealing a Bitcoin private key enables an adversary to steal their money.

Many ideas have come up on how to keep the user's private keys secret. The regular Bitcoin wallet software for PC operating systems encrypts the private keys using AES and only decrypts them when the user wants to create a transaction. This gives basic protection against PC malware for the time the user does not use his private key. But when the user wants to generate a transaction and enters his passphrase, the unencrypted keys are in the application memory, and thus are not protected against malware.

The next step is to use a dedicated PC for Bitcoin only, and keep it disconnected from the Internet. This prevents the PC from being infected with malware, and even in case it gets infected, the malware has a hard job to send the secret keys home to the attacker. As an even further step, some companies start producing special purpose hardware which is designed to manage keys and create transactions, but to never release the keys themselves.

The strength of the latter two settings comes from the fact that even if the device with the Bitcoin private keys on it is malicious, it cannot send any private information to attackers as long as the user makes sure that only transactions leave the device. This is supposedly achieved, if the device for example writes the transaction into a file, and the user copies the file using a portable storage medium.

But, as we will see later, even if the user actually ensures that only the transaction leaves the wallet device, there is still a problem: ECDSA. Bitcoin transactions contain ECDSA signatures. In the process of creating

a ECDSA signature, like in the classic DSA, the creator has to choose a random number. Adam Young and Moti Yung showed in (1997), how this random number can be used by a malicious implementation of DSA to leak private information. This can be done in such a way that only the attacker, who made the malicious implementation, can extract the secret from the signature, and that the distribution of malicious signatures is polynomially indistinguishable from legitimate signatures by anybody else.

This paper will outline how this attack can be used for ECDSA as well, and what security issues arise from that for applications of ECDSA like Bitcoin.

2   Related Attacks

A number of attacks on Bitcoin keys have been around in the wild as well as in research publications, e.g. in works by Nicolas Courtois and Filippo Valsorda(Valsorda 2014), (Courtois, Emirdag and Valsorda 2014). Those attacks are enabled by same properties in the ECDSA scheme as the attack presented in this paper: When generating a ECDSA signature, the user has to generate a new random number that has to be kept secret, because knowledge of this number enables an attacker to compute the private key.

The attacks outlined by Valsorda and Courtois (and succefully run by many attackers of unknown attackers on the actual Bitcoin network) make use of weak randomness. If a user's wallet software is reusing the same random number for generating signatures several times, attackers can compute the private key and steal their Bitcoins.

The attack outlined in this paper is different, because it does not make use of bad randomness, which would potentially be detected by attackers scanning the network. In this attack, a malicious wallet implementation is generation two of those random numbers in a way that the attacker can compute the private key from two signatures, while for everybody else (who does no have the attacker's private key), the signatures look random.

3   Definitions

In our considerations, the *user* is the person, who wants to use a publicly specified cryptosystem, e.g. ECDSA. The *attacker* is the person, who creates a malicious implementation which is used by the user. The intuition is that the attacker's impementation can differ from how the cryptosystem is specified, but that the inputs and outputs have to comply with the specification. In addition we don't want the user or any third party to be able to distinguish the outputs generated by the malicious implementation from the outputs generated by a specification-compliant implementation.

3.1   Kleptographic SETUP

We will use the definition of kleptographic setups given by Young and Yung in (1997).

Definition.   Let C be the black-box implementation of a cryptosystem with publicly known specification. A kleptographic *(regular) SETUP* (Secretly Embedded Trapdoor with Embedded Protection) is a modified algorithm C' such that:

1. The input of C' agrees with he public specifications of the input of C.

2. C' computes efficiently using the attacker's public encryption function E (and possibly other functions as well), contained within C'.

3. The attackers private decryption function D is not contained within C' and is known only by the attacker.

4. The output of C' agrees with the public specifications of the output of C. At the same time, it contains published bits (of the user's secret key) which are easily derivable by the attacker (the output can be generated during key generation or during system operation like message sending).

5. Furthermore, the output of C and C' are polynomially indistinguishable (as in Goldwasser, Micali 1984) to everyone except the attacker.

6. After the discovery of the specifics of the setup algorithm and after discovering its presence in the implementation (e.g. reverse-engineering of hardware tamper-proof device), users (except the attacker) cannot determine past (or future) keys.

4 A Kleptogram for Elliptic Curves

In this section, we will see how elliptic curves can be used to hide information in the choice of a random number like it is performed in ECDSA. This will be the central building block in our later attack.

Given an elliptic curve E. Let G be a point on E of order n. Let d be the attacker's private key and $Q = dG$ the corresponding public key. Let $\mathscr{R}$ be a cryptographically strong pseudo-random number generator with hidden seed. Without loss of generality, we assime that $\mathscr{R}$ outputs a value less than n. Let α, β, ω be fixed integer constants, with ω being odd.

4.1 Generation

The malicious implementation of a elliptic-curve cryptosystem generates two subsequent choices c_1, c_2 the following way:

First round.

1. Pick random $c_1 < n$.

2. Compute $M_1 = c_1 G$.

3. Store c_1 in non-volatile memory.

4. Output M_1.

Second round.

1. Pick random bit $t \in \{0, 1\}$.

2. Compute $Z = (c_1 - \omega t)G + (-\alpha c_1 - \beta)Q$.

3. Compute $c_2 = \mathscr{R}(Z)$.

4. Compute $M_2 = c_2 G$.

5. Output M_2.

4.2 Recovery

Now the attacker is able to extract the (secret) value of c_2 from M_1, M_2 and his private key d as follows:

1. Compute $R = \alpha M_1 + \beta G$.

2. Compute $Z_1 = M_1 - dR$.

3. If $M_2 = \mathscr{R}(Z_1)G$ then output $c_2 = \mathscr{R}(Z_1)$.

4. Compute $Z_2 = Z_1 - \omega G$.

5. If $M_2 = \mathscr{R}(Z_2)G$ then output $c_2 = \mathscr{R}(Z_2)$.

4.3 Indistinguishability

Values in the first round are chosen at random. An attacker wants the distribution of values from the second round to be indistinguishable from values generated at random like in the first round.

For this reason, the second round makes use of a seeded pseudo-random number generator. To make the distribution of the generator's output as hard to distinguish as possible, the attacker wants the number of potential values for Z as large as possible, ideally n different values.

The attacker can achieve this by tweaking the constants α, β and ω such that

$$G_1 = (-d\beta - \omega)G$$
$$G_2 = (-d\beta)G$$
$$G_3 = (1 - d\alpha)G$$

have preferably high orders, the ideal being n. In that case, Z and thus $c_2 = \mathscr{R}(Z)$ can take on any value up to n. Assuming that the output of $\mathscr{R}$ is polynomially indistinguishable from random numbers of equal distribution, we can follow that only the attacker himself can distinguish the two (by blindly applying his recovery procedure).

5 ECDSA and Bitcoin

In this section, we will see how the above construction can be used to create a SETUP for ECDSA.

5.1 Setup for ECDSA

Let's recall ECDSA. For an elliptic curve E with a generator point G of order n, an ECDSA user's private key is a number $d < n$. The public key is the corresponding curve point $Q = dG$. Let $\mathscr{H}$ be the cryptographic hash function used.

A signature for a message m is generated as follows:

1. Pick a random value $k < n$.

2. Compute $R = (r_1, r_2) = kG$.

3. Compute $r = r_1 \mod n$.

4. Compute $s = k^{-1}(\mathscr{H}(m) + dr) \mod n$.

5. Output $\sigma = (r, s)$

The signature is verified by another party as follows:

1. Compute $R' = (r_1', r_2') = s^{-1}\mathscr{H}(m)G + s^{-1}rQ$.

2. Compute $r' = r_1' \mod n$.

3. Accept the signature if and only if $r = r'$.

Note that a number k_i is chosen by the algorithm for each signature σ_i. A black-box implementation of ECDSA can substitute the values k_i, k_{i+1} in two consecutive signatures σ_i, σ_{i+1} by c_1, c_2 generated like in section 3. This enables the attacker to extract the value of $k_{i+1} = c_2$ from $\sigma_1 = (r_1, s_1)$, $\sigma_2 = (r_2, s_2)$ as described, because r_1, r_2 are the equivalent to m_1, m_2 respectively from section 3.

Now note that knowledge of the value k for a single signature σ already enables an attacker to compute the private key d, since from step 4 in the signature generation, it follows:

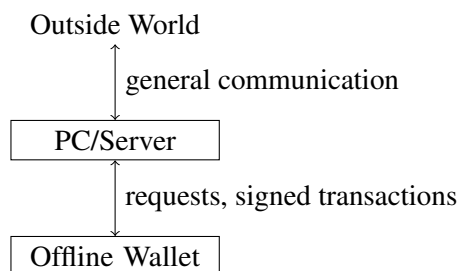$$d = (\mathscr{H}(m) - sk)r^{-1} \mod n$$

Figure 1. Bitcoin configuration with an offline wallet

5.2   Limitations

One limitation with this attack is that the attacker has to know in advance which curve and which generator point the user will choose to set up his implementation. Otherwise he cannot generate his own public-private key pair. This was a big limitation, when Young and Yung (1997) published the attack for classic Diffie-Hellman settings, because it was common that each user generates his own new group $\mathbb{Z}_p$ and a generator $g \in \mathbb{Z}_p^*$ for each application.

With elliptic curve cryptography, this limitation became less relevant. Since it is more expensive to generate a new strong pair of a curve and a generator point from scratch, elliptic curves are generated and analyzed by scientists, and only a few elliptic curves are published (with corresponding parameters including the generator point) by standardization bodies such as the U.S. governments' NIST. Some ECDSA applications even specify a particular single curve to be used, e.g. Bitcoin uses only the secp256k1[1] curve for ECDSA.

5.3   Bitcoin

Bitcoin is a decentralized payment system, which is implemented as a peer-to-peer network (Nakamoto 2008). In order to pay money to one another, users create little files named *transactions*. These transactions are broadcasted into the network and collected in a large log file called *blockchain*, which is maintained by all participating peers in a cooperative manner.

For our considerations we don't have to go further into how the blockchain is maintained in detail. We just have to keep in mind that the blockchain collects all valid transactions, and that the previous blockchain history is what gives the user a balance that he can spend.

To participate in the payment and receive money, a user creates a Bitcoin *address*. He creates an ECDSA public-private key pair.[2] To generate the address, the public key is hashed twice using the SHA256 algorithm.

Now a standard transaction is a file that specifies a number of inputs and outputs. For inputs, the transaction refers to a previous transaction that has charged one or more of the user's addresses with coins. For outputs, the transaction specifies one or more Bitcoin addresses owned by the receiver. To make the transaction valid, it has to satisfy the following conditions:

- The specified inputs must have a higher or equal value than the outputs are receiving.

- The transactions specified in the inputs must be part of the blockchain and not yet spent.

- The user has to prove that he owns the addresses from the input transactions by signing the transaction with the corresponding private keys.

---

[1] Published in 2000 by Certicom Research.
[2] Note that the secp256k1 curve is the only elliptic curve used in Bitcoin.

The last condition is the one that makes sure that only the owner of an address can spend the received coins, because only he knows the ECDSA private key.

Now note that since the signatures in a transaction are plain ECDSA signatures, the kleptographic SETUP from section 4 can just be applied in a straight forward manner. This means that a malicious programmer or hardware designer can implement a Bitcoin wallet in a way that leaks the secret key without any side channels using only two signatures. Note that since there might be more than one signature in a transaction, this can happen in a single transaction if two inputs associated with the same address are used. Nobody but the attacker is able to distinguish such a malicious transaction from a normal one.

5.4 Deterministic Choice of k

The kleptographic attack on ECDSA is very easy, because the value k has to be chosen during the signature creation. As we have seen, k has to be secret to prevent adversaries from extracting the secret key d. Only $R = kG$ is published with the signature.

Another slightly related security issue also arose from the fact that k has to be chosen by the signature algorithm. If two values k_1, k_2 in two different signatures have a known linear relationship $k_2 = ak_1 + b$ with $a, b \in \mathbb{Z}$, the private key d can be extracted from the two signatures without the knowledge of the values k_1, k_2, since it results in two linear equations with only d and k_1 unknown.

Because of this known attack, some proposals have been made on how to choose the number k deterministically, e.g. RFC6979 or the specification of EdDSA by Bernstein, Duif, Lange, Schwabe, Yang (2012). These proposals generate k deterministically using a cryptographic hash function with the message and the user's private key as inputs. The result is that the same user always creates the same signature for a given message. Note that it is crucial here to have the private-key part of the input. If k would be derived from the message alone, it would be public information and therefore useless.

This measure gives only limited protection against the kleptographic attack since it can only be verified using the private key. The whole point of a dedicated Bitcoin wallet is that the user wants to make sure that the private keys never get anywhere outside of it, which means that even the user himself cannot verify the signature using a second computer.

6 Potential Solutions

One potential counter-measure could be the following:

1. The signature device generates the signature with deterministic k as specified in RFC6979.

2. In addition to that, the device delivers a zero-knowledge proof that k was indeed generated as specified.

This is possible since it is known that zero-knowledge proofs exist for any NP statement. But with this solution, a new problem arises. The whole point of using an offline Bitcoin wallet is that it does not leak any information into the public except the legitimately generated transactions. If we let the device output a zero-knowledge proof in addition to that, this proof may introduce new ways for the adversary to leak information. This means that the zero-knowledge protocol for this application has to be chosen carefully. Another counter-measure would be to strictly not use any address more often than once. Although this way of using Bitcoin addresses is recommended already because of privacy considerations, there exist some use cases where this may not be feasible. For example, a public donation address for a charitable foundation is supposed to be used by multiple donors over and over again for a long period of time. In order to transfer the donations, a signature has to be created for each incoming payment.

A deterministic choice of k alone (as described in the last section) would not help much either, because knowledge of the private key is necessary to verify that k has indeed been chosen as specified. But there are still two advantages arising from using a deterministic method: First, it limits the choice of k if the user signs the exact same document twice, as k has to be identical given the same message and private key.

Outside World

↑

general communication

| PC/Server |

↑

requests, validated signatures

| Supervisor |

↑

interactive communication, unvalidated signatures
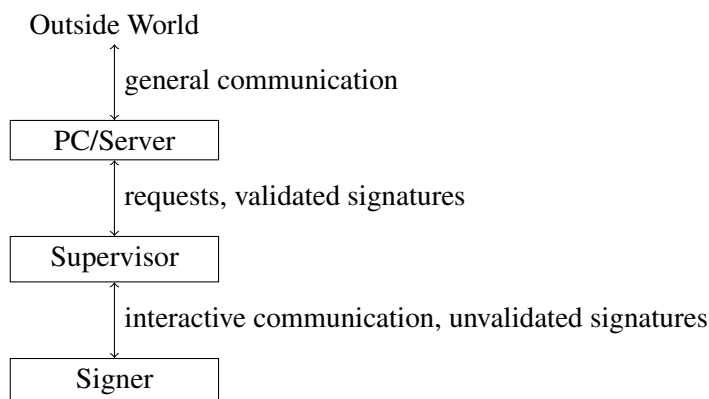
| Signer |

Figure 2.   Configuration with a supervisor

Secondly, it would enable the user to detect malicious signatures later, for example after a key has expired and it is safe to transfer the private key to another computer.

6.1   Interactively Generate k

The ability of the signer to leak information through his choice of k come from the fact that he is allowed to choose k freely. If an outside agent could force the signer to choose k from the equal distribution, that wouldn't be possible.

A common way to solve this cryptographic problem is an interactive protocol, where two parties choose a common random string. Such a protocol is arranged in a way that neither of the two parties can influence the resulting random string to his wishes, because both parties have to make their choice for their part before the other one's choice is revealed. This can be enforced using a commitment scheme.

In a blogpost[3] on firmcoin.com, the Certimix company describes such an interactive protocol that deals with the problem. In addition to the signing device (i.e. the offline wallet), we have a supervisor device that checks whether the signing is done properly. The protocol works as follows:

For an elliptic curve E with a generator point G of order n be $d < n$ the private key and the corresponding curve point $Q = dG$ the public key. Let $\mathscr{H}$ be the cryptographic hash function used.

A signature for a message m is generated as follows:

1. Supervisor and signer generate k together:

   a) The signer picks a random value $t < n$.

   b) The signer computes $T = tG$.

   c) The signer computes $h_T = \mathscr{H}(T)$.[4]

   d) The signer sends h_T to the supervisor.

   e) The supervisor picks a random value $u < n$.

   f) The supervisor sends u to the signer.

   g) The signer sends T to the user.

   h) The supervisor verifies that $h_T = \mathscr{H}(T)$.

   i) The signer computes $k = t \cdot u \mod n$.

2. The signer computes $R = (r_1, r_2) = kG$.

3. The signer computes $r = r_1 \mod n$.

---

[3] Blogpost from June 20, 2013, "No subliminal Channel", http://firmcoin.com/?p=52

[4] This is a commitment to T.

4. The signer computes $s = k^{-1}(\mathcal{H}(m) + dr) \mod n$.

5. The signer sends the resulting signature $\sigma = (r, s)$ to the supervisor.

6. The supervisor releases the signature after verifying that k was generated correctly:

    a) The supervisor computes $R' = (r'_1, r'_2) = uT$.

    b) The supervisor verifies that $r = r'_1 \mod n$ holds.

Note that only the supervisor has to communicate with the outside world. But there are still some things that we have to be aware of: Even though neither the signer nor the supervisor can manipulate the choice of k/r as published with the signature, the signer might still leak information to the supervisor through his choices of t_i. There are still advantages coming from the protocol: As the supervisor does not have to store any critical data in persistent memory, the security properties are different. In addition, as long as either one of signer and supervisor works correctly, the signature cannot leak information through the choice of k.

This protocol has the disadvantage that it requires a lot of interaction. But we can make this protocol more practical by performing a prearrangement step where a list of random numbers is generated in advance. Let E again be an elliptic curve with a generator point G of order n, be $d < n$ the private key and the corresponding curve point $Q = dG$ the public key. Let $\mathcal{H}$ be the cryptographic hash function used.

Prearrangement

1. The signer chooses a list of random numbers $t_1, \ldots, t_\ell$ with $t_i < n$ for each $i \leq \ell$.

2. For $i \leq \ell$, the signer computes $T_i = t_i G$ and $h_{T_i} = \mathcal{H}(T_i)$.

3. The signer stores the lust of choices $t_1, \ldots, t_\ell$ for later use.

4. The signer sends the list of hashes $h_{T_1}, \ldots, h_{T_\ell}$ to the supervisor.

5. The supervisor stores the list of hashes.

Signing

1. The supervisor generates a random number $u < n$.

2. The supervisor sends (m, u) to the signer.

3. The signer computes $k = ut_1 \mod n$ and $T_1 = t_1 G$, where t_1 is the first element from the list of choices.

4. The signer generates the ECDSA signature $\sigma = (r, s)$ using k as the nonce.

5. The signer sends (σ, T_1) to the supervisor and removes t_1 from the list of choices.

6. The supervisor computes $R' = (r'_1, r'_2) = uT_1$.

7. The supervisor verifies that $r = r'_1 \mod n$ and $h_{T_1} = \mathcal{T}_\infty$.

8. If successfully verified, the supervisor publishes the signature and removes h_{T_1} from the list of hashes.

If any verification step fails, the supervisor should cancel the protocol and alert the user.

7   Conclusion

Without a satisfying solution, there is only one conclusion to draw from this problem: Users cannot trust any implementation of ECDSA or Bitcoin, which they cannot fully verify.

Note that this does not only affect strict black-box implementations such as closed-source programs. A user with high security requirements (like an online shop that wants to accept Bitcoin payments on a large scale) would have to use an implementation, which can be verified by his own staff (or at least a reliable partner company). An hard-to-read implementation like OpenSSL may be insufficient, because the particular implementation of ECDSA may be hard to verify. Such implementations feature many

variations on the pure algorithm to improve performance (e.g. using CPU-dependent assembly language) or to harden the implementation against timing attacks). In addition, it is hard to verify in a large program like OpenSSL, which code is actually executed when you perform a certain operation.

A similar problem arises with embedded cryptographic chips like smartcards. Such devices are designed to never release the private keys and to make it hard for an outside analysis to read out secret data. The fact that the leak in the kleptographic attack is so well hidden makes it hard for a chip manufacturer to prove to the customers that the device does not leak any secret information in ECDSA signatures.

The paranoid among users would even have to compile the program themselves to be sure that the code their are reading really matches the code they are running. To verify that the executed code actually matches the source code is even harder for small embedded devices (like dedicated Bitcoin wallets or crpyto smartcards) than in the setting of an offline PC.

References

Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, "High-speed high-security signatures", *Journal of Cryptographic Engineering*, September 2012, Volume 2, Issue 2, pp 77-89, 2012, Springer-Verlag

Shafi Goldwasser, Silvio Micali, "Probabilistic Encryption", *Journal of Computer and System Sciences*, Volume 28, Issue 2, April 1984, Pages 270–299, 1984, Elsevier

Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System" (pseudonymously published in 2008)

Internet Engineering Task Force (IETF), "Request For Comments (RFC) 6979"

Adam Young, Moti Yung, "The Dark Side of "Black-Box" Cryptography or: Should We Trust Capstone?", *Advances in Cryptology — CRYPTO '96*, Lecture Notes in Computer Science Volume 1109, 1996, pp 89-103, 1996, Springer Berlin Heidelberg

Adam Young, Moti Yung, "The Prevalence of Kleptographic Attacks on Discrete-Log based Cryptosystems", *Advances in Cryptology — CRYPTO '97*, Lecture Notes in Computer Science Volume 1294, 1997, pp 264-276, 1997, Springer Berlin Heidelberg

Filippo Valsorda, "Exploiting ECDSA Failures in the Bitcoin Blockchain", *talk at HITB 2014*

Nicolas T. Courtois, Pinar Emirdag, Filippo Valsorda, "Private Key Recovery Combination Attacks: On Extreme Fragility of Popular Bitcoin Key Management, Wallet and Cold Storage Solutions in Presence of Poor RNG Events", *Cryptology ePrint Archive: Report 2014/848, IACR*