Kulkan Security

Blog

Capture the Coins – Bitcoin Challenge – Explained

On August 2014, Kulkan Security took a shot at the Bitcoin "Capture the coins" challenge, brought by coinspect.co. The challenge was put together for the Ekoparty 2014 Security Conference that takes place in Argentina once a year. This article takes you through a step by step process on how Kulkan Security approached the challenge, how the challenged was solved, as well as all tools and resources that were used.

The purpose of the article is for readers to understand both the Bitcoin challenge and its solution, but also to dive into a series of techniques that may come in handy in future Bitcoin Security scenarios. Although I'll do my best at sharing a clean and effective solution here, I guess there's no need to explain the amount of effort the challenge actually took and the level of insanity I was brought to while trying to solve this. Making mistakes and coming to the wrong conclusions is not only permitted but necessary – and I've done it enough times not to forget Bitcoin for some time. I'll bore you no more – let's begin.

I've structured this article in PARTS, and I'd advise reading them in order:

PART 0 – INTRO TO THE CHALLENGE

Coinspect shared the following address via twitter, that got retwitted by friends:

> *3GSgmPLShQpKGDbLaAXJ6t1dknu5NmY1eZ*

Carrying the following text: "The multisig wallet used to test the Copay exploit is broken for ever? Take the bitcoins if you can."

And through the official Ekoparty mailing list:

> *32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe*

stating in spanish the address permitted "earning some bitcoins"

And then came the following tweet:

> *..bitcoins still there and more coming. Best write up about the challenge gets VIP ticket to @ekoparty.32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe*

PART I – FINDING SOURCES OF INFORMATION

So, we start off with the following pieces of information:

- The attack requires exploitation of a flaw related to the following Coinspect advisory: "Copay wallet emptying vulnerability"
- To complete the challenge we must transfer Bitcoins from a victim's address to our own.

Searching online for the "Copay wallet emptying vulnerability", leads to the following links:

- http://blog.coinspect.co/copay-wallet-emptying-vulnerability – El advisory de Coinspect
- http://blog.bitpay.com/2014/07/28/copay-security-vulnerability-discovered.html – El aviso oficial de Copay/Bitpay

We go through the advisory contents, and look for additional resources on how Bitcoin transactions work:

- https://bitcoin.org/en/developer-guide
- https://en.bitcoin.it/wiki/Transactions
- https://en.bitcoin.it/wiki/Script
- https://en.bitcoin.it/wiki/Protocol_specification

Browsing around blockchain.info, we notice the site provides information on:

- Transaction history and address information (e.g.: https://blockchain.info/address/32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe)
- Transaction details (e.g.: https://blockchain.info/tx/6102bfd4bad33443bcb99765c0751b6b8e4e65f4db4e3b65324c5e9e3dac8132)
- Among other features. Some provided through an API: https://blockchain.info/api

PART II – UNDERSTANDING THE COPAY BUG AND SIGHASH TYPES

Coinspect makes a strong emphasis in different opportunities on how the "Wallet emptying" flaw discovered in Copay is strictly bound to the Bitcoin challenge. So we move forward and read the advisory (watch the video they made as well!.) Here's our explanation just in case you need help understanding the concepts discussed:

- The flaw discussed in the advisory affected Copay (https://copay.io) and got fixed right after away.

- The flaw involves multisignature wallets – where addresses have multiple owners who authorize each outgoing transaction.
- The flaw existed because Copay did not force a particular Hashtype (HASHTYPE__ALL) on transaction proposals; but rather used the one that was selected by the proposal originator (HASHTYPE__NONE, or HASHTYPE__SINGLE.)

Let's look at it using an example:

Imagine there's a Bank entity under which people or businesses may open accounts where to place their money. In order to get money OUT or IN their accounts, they use paper checks. Individual accounts have one single owner who authorizes and signs checks. Business accounts have more than 1 owner, and the Bank provides security by requiring a minimum amount of owners to authorize/sign each check prior to considering them valid (e.g.: 2 out of 3 owners..4 out of 6.. etc.) Such feature provides security against one single account owner clearing the account savings with no one else from the company even knowing what was going on.

The Bank provides one extra feature or benefit. More than one account (Individual and/or Business) may agree to write a check together, to send money to one or more entities. For example, should two companies share office space – they could write a check under the Landlord's name, paying half each. They could even include the Internet service provider and a payment to the phone company in the same check, and specify different amounts for each receiving end. Cool, huh!

Now, let's discuss paper checks! There are different ways one may sign a paper check. Each business, or individual, selects a particular way the check is going to get signed. In a business check, all owners of the account must agree to sign the same way.

The different types of signatures / ways of signing a check are:

SIGHASH_ALL – The front of the check is signed. Ink goes all over the recipient name, as well as the amount of money being received. This is the most secure way to sign the check. Altering the recipient name as well as the amount would require being able to erase someone's signature, and then re-doing t on top of the altered information. Impossible!

SIGHASH_SINGLE – The front of the check is signed, similarly to SIGHASH_ALL but with one particular characteristic. Given multiple recipients, the Ink will only cover 1 of their names and amount. Which one? The one that's right in line/aligned with the individual/business account in the paper check. That's why it's called SINGLE. In a paper check with 1 recipient, that's ok. However a check with more recipients than senders.. risky situation!

SIGHASH_NONE – Signatures go in the back of the check! No ink gets to cover name and amount of ANY recipients, which go in the front side of the check. Signatures are still validated by the Bank though to make sure it was written by account owners, but they provide almost no security whatsoever. This is the closest it gets to a Blank check. The horror!

SIGHASH_ANYONECANPAY – By signing this way, account owners leave some space for other businesses or individual account owners to join forces and add funds to the check. This makes sense in a Donation check, for instance. Don't worry about this type of signature though – won't apply to this challenge.

A paper check ends up being secure or insecure based on the way its been signed once all account owners that participate in the check have finished signing. Makes sense?

Coinspect's advisory considers a Business account scenario, also known as a multi signature wallet. Account owners assume all checks are being signed with SIGHASH_ALL, whereas that may not be the case at all! One of the account owners turning Evil may submit a paper check proposal to the others forcing SIGHASH_SINGLE or SIGHASH_NONE, without anyone really knowing what's happening. Here's what could happen:

1) If SIGHASH_NONE is used, the evil owner (or anyone else, really) may alter 1 or more recipient names as well as their amounts. The check ends up being almost a Blank check. The evil owner is really happy.

2) If SIGHASH_SINGLE is used, and the amount of recipients exceeds 1 (one), the evil owner may change one (1) or more recipients and their corresponding amounts.

In other words; Copay allowed whoever originated the transaction proposal to decide which HashType was going to be used by everyone signing. The right thing to do would have been to force SIGHASH_ALL, disregarding what the proposal originator sent. Good news is that Copay is now forcing it!

By now we should have understood what HashTypes are for, as well as what was the security problem that Coinspect identified in Copay. But how do we use what we learnt? Let's move to PART III.

PART III – BASIC CONCEPTS & PYCOIN

If you're here then I assume you've read all previously mentioned sources of info, and hence you understand:

- In Bitcoin, coins are transferred from 1 or more Inputs, to 1 or more Outputs.
- What an unspent OUTPUT is (A transaction[tx] id + an index which holds coins that haven't been used in an INPUT yet.)
- That Bitcoin uses its own SCRIPT language to validate signatures, to hash, to creates copies of data, etc.
- That Bitcoin's SCRIPT language is called ScriptPubKey if included in the OUTPUT section of a transaction..
- That Bitcoin's SCRIPT language is called ScriptSig if included in the INPUT section of a transaction.
- That ScriptPubKey exists in order to establish conditions that have to be met by whoever references said OUTPUT as INPUT.

Because Copay's bug is connected to HashTypes, and the way signatures are created in a Bitcoin transaction, then we're going to need to be able to view transactions in raw format, or close to raw. Googling for ways to get Transaction data leads to multiple web-based options; but we want a command-line based tool, so we find:

https://github.com/richardkiss/pycoin

Getting and installing pyCoin is a trivial process [python setup.py install] – it provides a set of interesting command-line tools, but right now the one we care about is named "tx". Jump right to its manual available at:

https://github.com/richardkiss/pycoin/blob/master/COMMAND-LINE-TOOLS.md

But.. what transaction do we focus on? And which of the two addresses that Coinspect provided is the one? Let's answer those questions in PART IV!

PART IV – FINDING THE RIGHT BT ADDRESS
Coinspect provided 2 (two) different Bitcoin addresses for the challenge. Which of the two do we use? Both? None? The goal of the challenge is to transfer Bitcoins from an address owned by Coinspect (we hope, you never know!) to one that we own. In order to accomplish such a thing, we would first need to identify transactions that hold unspent OUTPUTs. We assume we look for unspent and not spent – for the latter would mean total Mayhem in the Bitcoin network.

Let's load both addresses via Blockchain.info:

https://blockchain.info/address/3GSgmPLShQpKGDbLaAXJ6t1dknu5NmY1eZ
https://blockchain.info/address/32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe

and look for unspent bling bling. We can also use pyCoin for that purpose, and its "fetch_unspent" tool:

As seen above, the first Bitcoin address returned nothing, not a single dime. The second one did though! meaning that chances are we found our victim (32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe)

Disregarding how (blockchain.info/fetch_unspent/etc.) we've managed to find the source address for the funds that we're going to "steal", we notice the transaction with an unspent OUTPUT is, in this case:

9a5b462b6ecae93fc091cb9d3402c8c1e053ae30720150509c029e9e92602bd1.

Excellent – so far we believe we have:

- The victim's Bitcoin address from which we'll be "stealing" coins: 32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe
- The transaction which holds in one of its outputs our victim's address with unspent coins: 9a5b462b6ecae93fc091cb9d3402c8c1e053ae30720150509c029e9e92602bd1

But.. going back to what we learnt in PART II, what we really need in order to exploit the flaw is an **insecure** check or transaction holding unspent coins. A check or transaction where everyone involved has signed using a HashType of either SIGHASH_NONE, or SIGHASH_SINGLE.

For that purpose, we go back to blockchain.info y and search through our victim's address transaction history: http://blockchain.info/address/32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe

A transaction with id "6102bfd4bad33443bcb99765c0751b6b8e4e65f4db4e3b65324c5e9e3dac8132" calls our attention. It has three (3) INPUT transactions bound to the same address, our victim (back to the example from PART II: this would be 3 different Business accounts owned by the same Company): https://blockchain.info/tx/6102bfd4bad33443bcb99765c0751b6b8e4e65f4db4e3b65324c5e9e3dac8132

From the information that blockchain shows us, it isn't clear though which HashType has been used on each INPUT. So we take charge, and make use of pyCoin's "tx" command in order to get a Hex-encoded string of the transaction contents:

The "tx" output includes plenty of info, but we want the dump of bytes in the end. In PART V we get to break apart that chunk of bytes in order to understand, among other things, which HashTypes have been used in the transaction.

PART V – UNDERSTANDING A RAW TX AND LOOKING FOR HASHTYPES

Back in PART IV we used pyCoin's "tx" command to dump the contents of a transaction encoded in hexadecimal. There are multiple tools online that can be used to parse the Hex dump, such as: https://blockchain.info/decode-tx, but we feel a lot closer to the Bitcoin protocol if done manually. Having read Bitcoin documentation listed in PART I, we break the transaction in the following pieces:

01000000 [Version]
03 [Total amount of Inputs included in the transaction]
77b5aa00744a7fee0754bcacb5aa5c46c8bd03ad3c1fd3f1ad5dacc3ca402aec [First TX – address inverted: ec..77]
01000000 [index within the TX. Starts in 0]
fdff00
00 [Marks the start of signatures]
48 [Size of the first signature]]
3045022100e5d7c59ea1fb5d0285e755dfc09634e1e3af36d12950b9b5d5f92b13602
1b3d202202c181129443b08dcfb8d9ced3018
7186c57c96f9cdb3f3914e0798682ea35d2b
03 [HashType! Found it! It's the last byte after each signature.]
49 [Size of second signature]
3046022100e1f8dbad16926cfa3bf61b66e23b3846323dcabf6c75748bcfad762fc50b
faf402210081d955160b5f8d2b9d09d8838a2
cf61f5055009d9031e0e106e19ebab234d949
03 [HashType used in second signature]
4c [OP_PUSHDATA1]
69 [OP_VERIFY]
52 [OP_2 – Marks the start of Public Keys. Being "_2" this means: 'a minimum of 2 signatures..']
21 [Public key size]
023927b5cd7facefa7b85d02f73d1e1632b3aaf8dd15d4f9f359e37e39f0561196
21 [Public key size]
03d2c0e82979b8aba4591fe39cffbf255b3b9c67b3d24f94de79c5013420c67b80
21 [Public key size]

03ec010970aae2e3d75eef0b44eaa31d7a0d13392513cd0614ff1c136b3b1020df

53 [OP_3 – End of Public Keys, and being "_3" this means: '..out of 3 pubkeys']

ae [OP_CHECKMULTISIG – Validate signatures!]

ffffffff [Delimiter. This is the end of the first Input]

06198d754e425f17f7d749dd8c2f03538159621d0c69ad1d71c7768b22fdeb30

[Second INPUT TX, reverse order]

00000000 [index within the TX]

fdfe00

00 [Marks the start of signatures]

49 [Signature size, 0x49]

3046022100b762c6946eb7f349e53cb5b7f129375bfee75a1b5e042c2e2f26233af106

f54c022100be0e0f8838681ef439f931709c93

5fbff4cca1949f4b9ad4ef143857180c0698

03 [HashType]

47 [Signature size, 0x47]

30440220275e7e34a066fdaacfd63030bca9e234f4380b6ae2c0f20e6ffe0dd6ddd7e

24702202f62435cc36d4ada9586d605faa039e9f5a1e2684631152c96d8dd5850171

825

03 [HashType]

4c [OP_PUSHDATA1]

69 [OP_VERIFY]

52 [OP_2: 'a minimum of 2 signatures..']

21 [Size public key #1]

023927b5cd7facefa7b85d02f73d1e1632b3aaf8dd15d4f9f359e37e39f0561196

21 [Size public key #2]

03d2c0e82979b8aba4591fe39cffbf255b3b9c67b3d24f94de79c5013420c67b80

21 [Size public key #3]

03ec010970aae2e3d75eef0b44eaa31d7a0d13392513cd0614ff1c136b3b1020df

53 [OP_3: '..out of 3 pubkeys']

ae [OP_CHECKMULTISIG – Validate signatures!]

ffffffff [Delimiter]

c50f192f5d21fd1ea12488cac94536f7428ad9aebf5ee7a106e26a80d47637c2 [Third input TX, reverse order]

00000000 [Index within the TX]

fdfd00

00 [Mark start of signatures]

48 [Size of signature, 0x48]

3045022100dfcfafcea73d83e1c54d444a19fb30d17317f922c19e2ff92dcda65ad09cb
a24022001e7a805c5672c49b222c5f2f1e67bb
01f87215fb69df184e7c16f66c1f87c29
03 [HashType]
47 [Size of signature, 0x47]
304402204a657ab8358a2edb8fd5ed8a45f846989a43655d2e8f80566b385b8f5a7
0dab402207362f870ce40f942437d43b6b993
43419b14fb18fa69bee801d696a39b3410b8
03 [HashType]
4c [OP_PUSHDATA1]
69 [OP_VERIFY]
52 [OP_2: 'a minimum of 2 signatures..']
21 [Size of public key #1]
023927b5cd7facefa7b85d02f73d1e1632b3aaf8dd15d4f9f359e37e39f0561196
21 [Size of public key #2]
03d2c0e82979b8aba4591fe39cffbf255b3b9c67b3d24f94de79c5013420c67b80
21 [Size of public key #3]
03ec010970aae2e3d75eef0b44eaa31d7a0d13392513cd0614ff1c136b3b1020df
53 [OP_3: '..out of 3 keys']
ae [OP_CHECKMULTISIG – Validate signatures!]
ffffffff [Delimiter: Here comes the OUTPUT section]
02 [Amount of outputs: 2]
204e000000000000 [Amount of Satoshis being received by the 1st output: 0x4e20 == 20000]
19
76a9 [OP_DUP, OP_HASH160]
14 [Size of output address/hash]
123a7dc5daca3b4a6ebb5ca788b60405dcee84ee
88ac [OP_EQUALVERIFY, OP_CHECKSIG]
e075090000000000 [Amount of Satoshis being received by the 2nd output: 0x9075e0 == 620000]
17
a9 [OP_HASH160]
14 [Size of 2nd output address/hash]
1f420aab25a69dc3a192ffc7f937fc82bb57afb6
87 [OP_EQUAL]
00000000

From our manual TX parsing process we've found:

- There are 3 inputs.
- Each of the inputs is carrying 2 signatures, out of 3.
- Each signature has a HashType 0x03 [SIGHASH_SINGLE]
- There are 2 outputs.

PART VI – WE HAVE FOUND A TX WITH SIGHASH_SINGLE
Discovered during our manual TX contents inspection in PART V, we've found all HashTypes used by each signature in the transaction. All of the HashTypes happen to be 0x03 in value, and according to the OP_CHECKSIG documentation (https://en.bitcoin.it/wiki/OP_CHECKSIG), this means we have SIGHASH_SINGLE signatures.

Thinking back of what we've learnt from the Copay bug, we realize each INPUT is only signing 1 (one) OUTPUT; the OUTPUT that shares the same Index number as the INPUT within the transaction. Being the INPUTs:

INPUT 1 –
ec2a40cac3ac5dadf1d31f3cad03bdc8465caab5acbc5407ee7f4a7400aab577:1
INPUT 2 –
30ebfd228b76c7711dad690c1d62598153032f8cdd49d7f7175f424e758d1906:0
INPUT 3 –
c23776d4806ae206a1e75ebfaed98a42f73645c9ca8824a11efd215d2f190fc5:0

and the outputs:

OUTPUT 1– 12fPEVz7t6tpvUUCzMT1iCCmtKcNqpUhv1 [20000 Satoshis == 0.0002 BTC]
OUTPUT 2 – 34YHz9Suwcfi1DsgvMwQVVyy1k5dX5oN6B [620000 Satoshis == 0.0062 BTC]

Then INPUT 1, signs OUTPUT 1. And INPUT 2, signs OUTPUT 2.

But wait.. **INPUT 3**? We had discussed a scenario with more OUTPUTs than INPUTs before, but what happens in case there are more INPUTs than OUTPUTs? Does this even work? Why is this happening to us?! We've been good to Bitcoin this far – why isn't Bitcoin being good to us?! We say **enough!** and call it for the day – we sleep poorly.

The following morning while reading the so many technical documents we've read so many times before, we find a particular Note in one of them (https://en.bitcoin.it/wiki/OP_CHECKSIG) A Note on SIGHASH_SINGLE:

"Note: The transaction that uses SIGHASH_SINGLE type of signature should not have more inputs than outputs. However if it does (because of the pre-existing implementation), it shall not be rejected, but instead for every "illegal" input (meaning: an input that has an index bigger than the maximum output index) the node should still verify it, though assuming the hash of 0000000000000000000000000000000000000000000000000000000000000001"

> *We bang our head against the desk. We know we're close, but we still can't connect the dots.*

PART VII – ANOTHER POINT OF VIEW HELPS CONNECT THE DOTS
In order to make progress, we change the way we approach the challenge. We now think of what the final malicious TX that we plan to push to the Bitcoin network should be like, in order for it to get validated and for us to "steal" unspent balance from 32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe.

Let's look at the basic ideal plan:

- For INPUT, we use a transaction carrying unspent balance from 32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe
- For OUTPUT, our own address.

We would need to be able to generate 32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe's signature in order to create a fresh INPUT coming from that address. But for such a thing we would need the corresponding private key, and we've agreed not to start sending out fishy looking e-mails to Coinspect. Furthermore, there's nothing cool about solving the challenge by owning the INPUT's private key – we want to use what we've learnt from the Copay bug!

But if we can't generate a signature for 32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe – could we just reuse one from before? We feel something moving in our brain. We think: If a transaction was created insecurely by using SIGHASH_NONE or SIGHASH_SINGLE – does that mean we could just use an INPUT from that TX and find a way to forge our own OUTPUT in a new transaction? How?!

1. For INPUT we reference a transaction with unspent balance from 32GkPB9XjMAELR4Q2Hr31Jdz2tntY18zCe, and where the signature should go, we just copy the one we obtained from the transaction that used HashType 0x03.
2. For OUTPUT, our own address.

All done? No! **Won't work**. There's 1 (one) input, and 1 (one) output. We've read before on how SIGHASH_SINGLE signs the OUTPUT that shares the same index as the INPUT. So again, that means we would need to own the private key for that INPUT in order for the signature to include our address in the OUTPUT – and believe me, no matter what we do, Coinspect won't give that away easily!

So let's just add more INPUTs!

> *voilà! The sky clears. The sun comes out! Nonetheless we have little idea about what's going on outside – we've been inside, with the computer, for an extended period of time.*

If we add more INPUTs and leave just this ONE output, then any INPUTs with SIGHASH_SINGLE that have NO output (what we explained in PART 6) in their same index will still VALIDATE and work – this is Bitcoin Law. So:

INPUT #1 – An extra INPUT that we add. If we added it as INPUT #2, then we'd be back to the same problem. And we can't add someone else's unspent TX – due to the fact that we don't own someone else's private key (let's assume..) in order to sign the OUTPUT. So what do we add as INPUT #1?

INPUT #2 – This is going to be the TX with unspent balance. The victim's address from where we'll be taking the Bitcoins.

OUTPUT #1 – Our own address! This is where we target to transfer all the coins that, by now, we think we even deserve!

So among the options that we have for INPUT #1, there are:

1- Finding a TX with unspent balance, from someone who's mistakenly (or with intention) created an insecure Transaction, creating signatures using SIGHASH_NONE as a HashType. A Bit hard to find last minute I'd say, but we could definitely try..

2- Using a TX with unspent balance that WE own. As long as we haven't misplaced our own private key that is. Let's just sign the OUTPUT ourselves!

We chose option #2.

In order to move forward, you will need an address that you own along with its private key. You will also need to have an unspent TX pointing to your address as OUTPUT. I used pyCoin's "ku" command to create a new address, to which I transferred a small amount of BTC that was left as "unspent". Don't spend too much money on this – mistakes could be quite expensive!

PART VIII – CREATING THE MALICIOUS TRANSACTION

We had a breakthrough in PART VII – we now know what to do. So, how do we put together a TX? Assembling it by hand can be lots of fun – but I'll show you here how easy it gets by using the "tx" command one again. Don't worry though – we'll get to manually edit a small portion of it.

1- Search for transactions with unspent balance that belong to our own address:

2- Search for transactions with unspent balance that belong to the victim's address – we know HashType 0x03 was used:

3- Create an unsigned transaction (a TX that is not signed simply holds the ScriptPubKey contents in the segment where the signature should be – and will NOT be considered valid until all signatures are in place) by using the "tx" command along with a list of INPUT transactions, their indexes, amounts, and the destination address – all delimited by a space character:

4- Let's check if the TX that we just created is valid:

We notice from above that the "tx" command even calculated automatically the sum of all INPUT amounts and the transaction fee (had we known this back when we started creating the TXs by hand!) and we also see a "BAD SIG" label next to each INPUT. Well that makes sense! none of the INPUTs have been signed just yet – we haven't signed ours – and we haven't yet stolen the signature from the victim's insecure TX with HashType 03.

4- Let's sign the TX using OUR private key. This is going to sign OUR input transaction. For signing, the "tx" command needs our private key in Wif format – which we have obviously not included in the example:

```
legendary:Desktop lucas$ tx tx-unsigned.hex XXXXX-CLAVE-PRIVADA-FORMATO-WIF-XXXXXXXXX -o tx-signed.hex

signing...
warning: 1 TxIn items still unsigned
all incoming transaction values validated
legendary:Desktop lucas$
```

Let's now go back and check the TX again to make sure our signature works. You will notice our INPUT now states "sig ok". One more signature to go:

5- Let's copy from PART V's hex dump any of the INPUT signatures carrying a HashType 0x03, including all signatures within the INPUT as well as all public keys. We copy from the first byte that follows the transaction INPUT index, all until the last byte prior to the INPUT delimiter (0xffffffff.) For example, an extracted chunk would look like:

```
fdff0000483045022100e5d7c59ea1fb5d0285e755dfc09634e1e3af36d12950b9b5d
5f92b136021b3d202202c181129443b08dcfb8
d9ced30187186c57c96f9cdb3f3914e0798682ea35d2b03493046022100e1f8dbad16
926cfa3bf61b66e23b3846323dcabf6c75748b
cfad762fc50bfaf402210081d955160b5f8d2b9d09d8838a2cf61f5055009d9031e0e1
06e19ebab234d949034c695221023927b5cd
7facefa7b85d02f73d1e1632b3aaf8dd15d4f9f359e37e39f05611962103d2c0e82979b
8aba4591fe39cffbf255b3b9c67b3d24f94de79
c5013420c67b802103ec010970aae2e3d75eef0b44eaa31d7a0d13392513cd0614ff1c1
36b3b1020df53ae
```

Let’s now edit the “tx-signed.hex” file that we just signed using the “tx” command. Replace the 0x00 byte right before the last 0xffffffff delimiter (it marks the start of the OUTPUT section – it may move though depending on the amount of INPUTs that we include) with the signature chunk that we stole from the vulnerable victim’s transaction and let’s check once again with the “tx” command to see what we’ve accomplished:

> *We did it! We’ve managed to create a valid TX using unspent balance originated from the victim’s address! Party time!*

PART IX – PUSHING THE TX TO THE BITCOIN NETWORK AND CASHING OUR BLING BLING

All that’s left to do is to push the malicious valid transaction to the Bitcoin network. There are multiple tools we could use, e.g.:

https://blockchain.info/pushtx
http://mec.blockr.io/tx/push

But the one that helped the most for this purpose (Others may not get along with multisignature transactions), was “sx” (https://sx.dyne.org) and its “sendtxp2p" parameter:

All done! Now wait for the TX to get validated in the network – once it gets validated, we get our money.

Thanks for reading – and I seriously hope you've managed to enjoy such a boring and long process that was actually so fun to us!

PART X – LINKS/TOOLS/SOURCES OF INFORMATION

- Coinspect.co – Bitcoin Security services
- http://www.ekoparty.org – Ekoparty Security Conference
- https://sx.dyne.org – sx tools for the Bitcoin cryptocurrency
- https://github.com/richardkiss/pycoin/ – pyCoin
- http://blog.coinspect.co/copay-wallet-emptying-vulnerability – Copay wallet emptying vulnerability advisory
- https://bitcoin.org/en/developer-guide
- https://en.bitcoin.it/wiki/Transactions
- https://en.bitcoin.it/wiki/Script
- https://en.bitcoin.it/wiki/Protocol_specification
- http://satoshi.24ex.com – BTC/mBTC/uBTC/Satoshi converter

Lucas / August 8, 2014