CSYE7374 HW4 Instructor: Dr. Handan Liu 2020-08-07

EOSIO provides a myriad of inbuilt accounts and permissions that allow cybersecurity to be built into its chain features, from the ground up. To protect themselves and their users’ funds, it behooves smart contract developers to learn about the true potential of these features and apply them to varying contexts in the ecosystem

# What’s important to consider when setting up smart contract security and key management?

EOSIO provides higher degrees of usability and security than traditionally thought possible on smart contract platforms through its native ability to collect signatures on-chain for multisig proposals

Understanding multisig schemes and permissions allow developers to utilize the wide range of security configurations that EOSIO affords in helping protect smart contracts that act custody users’ funds.

This guide also benefits users that actively interact with EOSIO dapps by providing an understanding of EOS account security in layman’s terms.

Different levels of security

|  |  |  |
| --- | --- | --- |
| Security Level | Technique | Relative Security |
| Maximum Security | **linkauth** using **sibling streams** for segregated action calls | 9 / 10 |
| Very Strong Security | **nested linkauth** with dedicated keys for specified action call | 8 / 10 |
| Strong Security | active-owner multisig with **single-key time-delay** | 7 / 10 |
| Medium / Strong Security | active-owner multisig **proposals** and **collectsigs** via eosio accounts | 6 / 10 |
| Median Security | active-owner **multi-vault** multi-key signatures via **passing** .json files | 5 / 10 |
| Medium / Weak Security | active-owner **single-vault** but requiring multiple signatures | 4 / 10 |
| Weak Security | active-owner multisig with **single-key time-delay** | 3 / 10 |
| Weaker Security | all contract actions available to single key with **active** permission | 2 / 10 |
| Weakest Security | all contract actions available to single key with **owner** permission | 1 / 10 |

**Weakest Security:** all contract actions available to single key with owner permission

The default setting where the same private key is used for your Owner and Active permissions. This is the **worst** eosio key configuration as it involves using your EOSIO account with the default owner key mapping for all universal actions.

The Owner permission is the "root access" to your EOSIO account. Only a few transactions require this authority. Access to the private key for the Owner permission allows you to perform any function on your account so it is suggested that you keep your Owner Key in cold storage and not shared with anyone.  However, with this default configuration, even when you choose to use your **Active Permission** to sign transactions, your **Owner Permission** is constantly exposed as it shares the same underlying key.

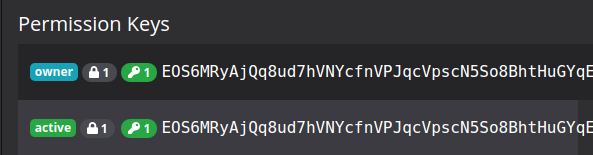


Figure 1.

**Recommendation:** Ideally, if you are ever required to use your Owner key, then it should be immediately reset. Consider it a one-time-use key.

**Weaker Security:** all contract actions available to single key with active permission

This is a slightly improved model as it involves separately generating a secondary set of key-pair and then assigning the public-key of that key-pair to the Active key of that account. Once this is set, the user can log out from their owner's permission and start accessing the same account, strictly with the newly set active keypair, with its more limited privileges. With proper assignment, one can use their EOSIO account with the more limited active-key privileges, while keeping their original owner keypair unexposed. Moreover, accounts accessed via the Active key cannot modify owner permissions.  

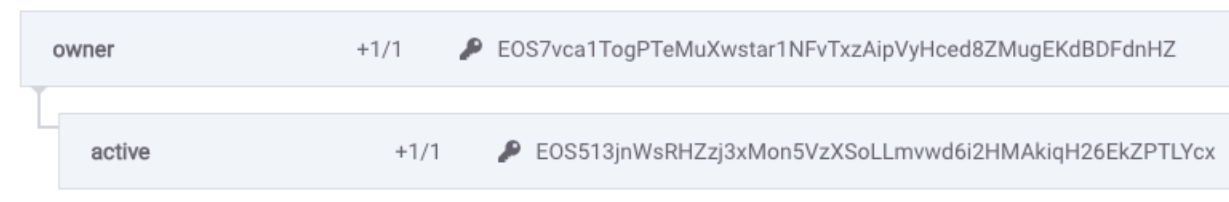



Figure 2.

To achieve this, Bob would have to craft a new time delay permission using a timedelay.yaml file as shown in **Figure 3** with a **wait\_sec** parameter of **90 seconds**.

---

threshold: 2

keys:

- key: EOS7vca1TogPTeMuXwstart1NFvTxzAipVyHced8ZMugEKdBDFdnHZ

weight: 1

permission: active

waits:

- wait\_sec: 2500100

weight: 1

permission: active

---

Figure 3. **timedelay**.yaml

To **activate** the timedelay permission, Bob would then need to use the **eosc** system **updateauth** command, to change his active permission to this new permission.

**$ eosc system updateauth bobacct active owner timedelay.yaml**

Why is this better than simple active / owner permission separation?

Occasionally, all humans may mess up. Separation of account permissions, may dilute the severity of an accidental action, such as a table deletion, but it does not allow the undoing of an action that is legal and already taken place. Such retrospective remediation is only possible with the aforementioned time delays.

**However, this setup, in terms of security design, is still very much a band aid solution as the strength of the security depends heavily on the length of the delay, how well you track it, and how responsive you are!**



Usage

To send a valid transaction that meets the time delay criterion, Bob would need to use the –delay-sec flag to specify a time delay **equal to or greater than** the required 90 seconds. Let’s say he wants to pay for his $1,750 lease next month. He could then schedule a transaction with a significant delay in seconds that would surpass the minimum required threshold:

**$ eosc transfer bobacct landlord 618.3745 --delay-sec 2600801**

Security Use Case

These are ideal for transactions involving high-value accounts, which may be the target of hackers. Forced time delays provides the principle of **security-in-depth**: i.e. if a hacker obtains sensitive keys, the owner, Bob is afforded some time to review the chain of events and issue a cancellation of transactions he deems to be malicious. Let’s assume **Figure 13.** is a malicious transaction that Bob wants to undo.

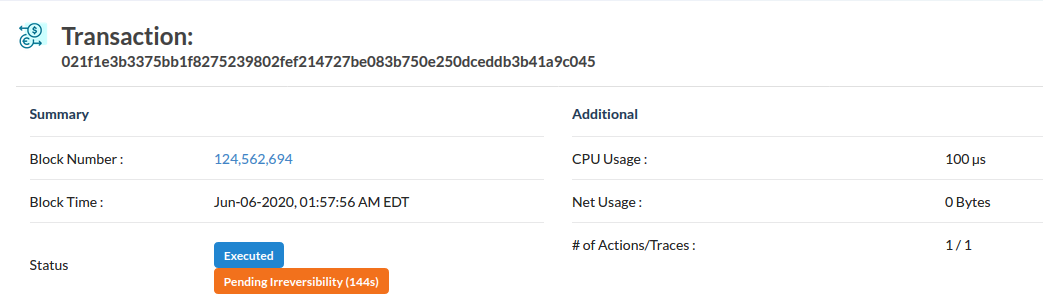


Figure 4.

Bob can then use his higher-level **owner** permission to alter the compromised lower-level permission that corresponds to the compromised key, effectively locking out the hacker. To do this, Bob would use the following command, with his **owner** permission flag (to override the otherwise present time-delay condition of the **active permission**), on the malicious transaction id:

**$ eosc tx cancel bobacct 021f1e3b3375bb1f8275239802fef214727be083b750e250dceddb3b41a 9c045 --permission bobacct@owner**



**Tip:**

Users should always subscribe to services like [EOS Authority](https://eosauthority.com/) or [dfuse](https://dfuse.io/?__hstc=183378681.e7b1974756854ed9cc955955e63fdf6e.1591216407477.1591418168748.1591421598430.8&__hssc=183378681.2.1591421598430&__hsfp=3163026173) to track, in real-time, pending transactions to their accounts and send them push notifications

**Weak Medium Security:** active-owner single-vault multikeysig

Originally used as a method of governance by which block producers are able to vote changes into EOSIO, multisig is useful for increasing the security of accounts and of smart contracts. Generally, multisig or multiple signatures require more than one signing key to execute a transaction. With EOSIO the system account, [**eosio.msig**](https://eosq.app/account/eosio.msig), handles the storing and execution of multisig **proposals** and their corresponding signatures on chain.

To increase the security of a given EOSIO account already utilizing active permissions (as discussed in the previous section), the user, say Bob, would hold one key, while someone else, say a trusted confidant, Alice, would hold **another key on his behalf**. Any actions that this account then took, would then require both keys **Key1** (key held by Bob) and **Key2** (key held by Alice) to confirm as an authorized transaction.

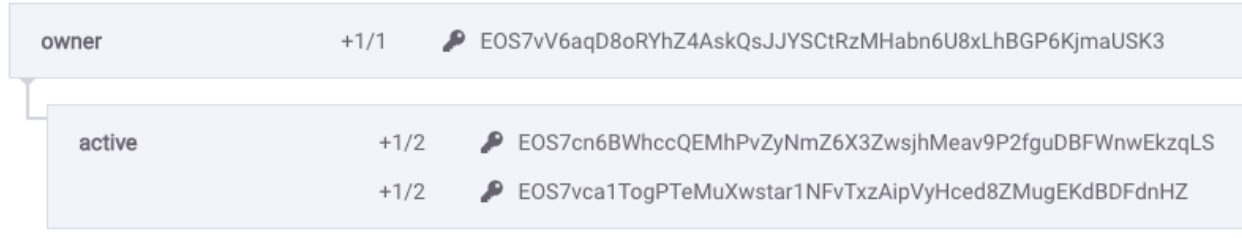


Figure 5.

**Figure 5** shows the **active** permission now broken into **2 separate keys**, Key1 (one belonging to Bob), and Key2 (in the possession of Alice). The +1/2 next to each key icon indicates that each key holds a weight of 1, and a total weight of 2 is **required**, for this account to sign off on any transaction requiring the active permission.

If the **Key2** protected by Alice, ending in **DFdnHZ** is compromised in any way, it alone would not allow the attacker from being able to successfully pass transactions. Any transactions signed by the attacker, on behalf of **Key2**, would not pass the multisig threshold of the active account, assuming Bob, who is privy to the compromise, refused to also sign the transaction in question, with **Key1**.





**Recommendation:**

Craft a multisig.yaml file as shown in Figure 4 and use the **eosc system updateauth** command as given below, to change your active permission to multisig.

---

threshold: 2

keys:

- key: EOS7cn6BWhccQEMhPvZyNmZ6X3ZwsjhMeav9P2fguDBFWnwEkzqLS

weight: 1

permission: active

- key: EOS7vca1TogPTeMuXwstart1NFvTxzAipVyHced8ZMugEKdBDFdnHZ

weight: 1

permission: active

---

Figure 6. **multisig**.yaml



**$ eosc system updateauth bobacct active owner multisig.yaml**

**Warning:**

The keys and accounts must be listed in alphanumerical order or you will receive an **"action validate exception"** error!

Fulfilling Multisig

To satisfy the **active** permission in this example, both **Key1** and **Key2** need to sign the transaction. In this weakest **single-vault** implementation of multisig, we assume that Bob and Alice are lazy, and have put both keys: **Key1** and **Key2**, in a single prearranged vault file. This file is then used to craft the two required signatures in one go.

**Warning:**

This arrangement of multisig has some resemblance to the cryptographic principle of security by obscurity, where the design focus is on maintaining secrecy as the main method of providing security to a system. From the outside looking in, this multisig arrangement may ward off would-be attackers when they inspect blockchain explorers like [**https://bloks.io/**](https://bloks.io/) as they consider it as multiple keys to compromise. This is true, if the **Key1** is exposed in some external system, as **Key2** would still be secure. However, the shared attack surface of the actual vault storing both keys, means that if the vault is exposed, then both keys are inherently compromised.



**Weakish Security:** active-owner multi-vault passing multikeysig

In this more secure implementation, Bob and Alice hold separate keys in **disparate** wallets, each secured in a different way (separate wallet etc).  In this case, let’s say Bob wants to make a transaction to pay $150 for his Amazon Cloud Practitioner certification using 53.5 EOSIO. Bob would have to **craft** a transaction that requires both him and Alice to sign-off on. He does this by providing a reasonable timeframe for Alice to receive and sign, say 20 minutes  (i.e. 1,200 seconds), and specifies this with the flag --expiration 1200. The command to craft the initial unsigned transaction would be something like this:

**$ eosc transfer bobacct awsacct 53.5 --expiration 1200 --skip-sign --write-transaction craft.json**

The transaction is now outputted to a **.json** file which Bob would need to sign himself, and then **pass to Alice**, for her to sign with **Key2**. The output unsigned file crafted by Bob would then look something like **Figure 7**:

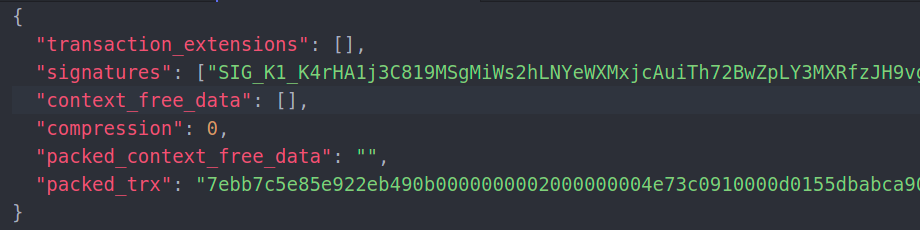


Figure 7. **craft**.json

Bob would then sign the crafted transaction with his first public key with the **eosc** command:

**$ eosc tx sign craft.json –offline-sign-key EOS7vca1TogPTeMuXwstart1NFvTxzAipVyHced8Zm ugEKdBDFdnHZ** **--write-transaction bob1sigcraft.json**

The result of Bob’s signature on the output file would result in something like **Figure 8**.

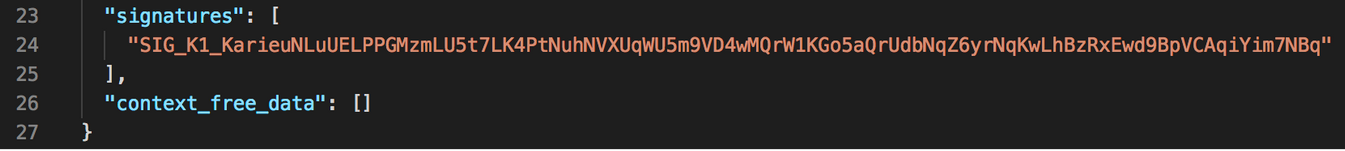


Figure 8. **bob1sig**craft.json

Bob would then pass this resultant **bobsigcraft.json** file over to Alice,, for her to sign (with the **Key2** key she is safeguarding). When Alice receives the file, she would sign it with much the same way, with the command:

**$ eosc tx sign bob1sigcraft.json --offline-sign-key EOS7cn6BWhccQEMhPvZyNmZ6X3ZwsjhMe av9P2fguDBFWnwEkzqLS --write-transaction bob2bob1sigcraft.json**

The result of this signing would lead to an **bob2bob1sigcraft.json** file that looks much like:

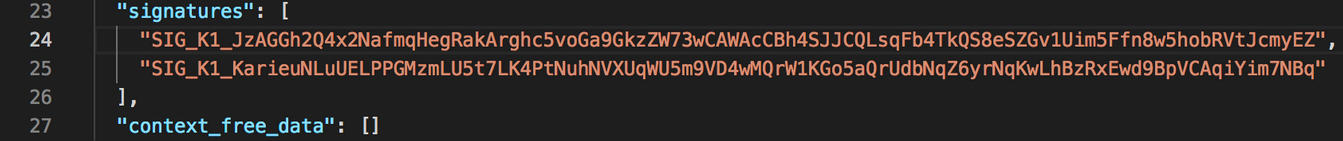


Figure 9. **bob2**bob1sigcraft.json

Once a multisig transaction has collected all its required approvals (in this case *2/2*), Alice can go ahead and submit the transaction to the network (or pass ***bob2bob1sigcraft.json***back to Bob) by executing the following command:

**eosc tx push bob2bob1sigcraft.json**

and then submit it to the network (all before the 20 minute time limit is up).

**Medium Security:** active-owner multi-account-sig

We now start to delve into more mature security implementations, by utilizing some of the inherent security afforded by EOSIO multisig. Instead of a single account (Bob) having multiple keys sign for a transaction, we have independent accounts. These accounts may be completely separate parties and may not even trust each other. In the arrangement below, Bob **(bobacct)** has to get the approval of two parties to get his transaction approved. These are his good friend Alice **(aliceacct)** who is simply safeguarding his key and his arch-nemesis landlord **(landlord)**, both of whom need to cooperate, to pay for the gas bill. This **multi-account-sig** permission arrangement structure would look as follows:

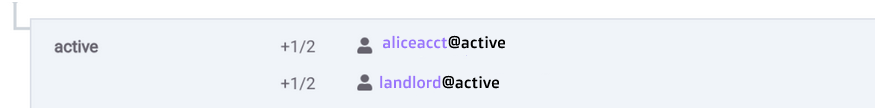


Figure 10. **bob2**bob1sigcraft.json

To satisfy the active permission, Bob needs to retrieve the signatures of Alice (a formality) and his **landlord** (more uncertain) to satisfy the active permissions of the accounts listed.

With the inbuilt **eosio.msig** contract, Bob no longer has to pass around files (like he did previously), especially since he doesn’t trust the **landlord** in the first place. Instead, the signatures are **requested** and collected into **temporary rows** within the eosio.msig contract **table**.

The first step for Bob to pay for the shared building gas bill (26.5 EOS), is to craft a transaction file similar to the previous example:

**$ eosc transfer bobacct councilacct 26.5 --expiration 2400 --skip-sign --write-transaction gasmsig.json**



Bob would then **multisig propose** this transaction, while requesting signatures from both **aliceacct** (Alice) and **landlord** (landlord) with the following command:

**$ eosc multisig propose bobacct collectsigs gasmsig.json --request aliceacct, landlord**

The outcome of this requested multisig proposal can be seen in **Figure 11** with the command:

**$ eosc multisig review bobacct collectsigs**

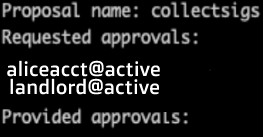


Figure 11.

Finally, the landlord, after some deliberation, also decides to go ahead and pay his share for the gas bill to the local council:

**$ eosc multisig approve bobacct collectsigs landlord**

The multisig transaction can now be executed on chain with the **exec** command as follows:

**$ eosc multisig exec bobacct collectsigs bobacct**

**Warning:**

**Any** account is able to call the exec command on a fully approved multisig transaction. The caller does not need to be part of the requested permissions. For example, the execution could have been called by the landlord or even Alice for that matter.

**$ eosc multisig exec bobacct collectsigs  aliceacct**

is equivalent to:

**$ eosc multisig exec bobacct collectsigs  landlord**

It may be prudent to plan out which account is in charge of ***executing*** the transaction, to ensure the calling account has sufficient CPU/NET bandwidth to execute the transaction. Although, there is no security threat rooting from failed calls due to insufficient resources, it is an unnecessary time overhead to retry transactions.



**Strong Security:** combined multi-account with time-delay

To build an even stronger security model, Bob can combine his own active key (**EOS6qiF84rDueuGuRYjJe6HwHN9hGPAUetpGGSVgBZp25zr34xcQ9**) with a time-delay and the added security of his secondary account vis-à-vis his trusted confidant, Alice (**aliceacct)** to achieve the permission model shown in **Figure 12.**

---

threshold: 2

keys:

- key: EOS6qiF84rDueuGuRYjJe6HwHN9hGPAUetpGGSVgBZp25zr34xcQ9

weight: 1

permission: active

waits:

- wait\_sec: 45

weight: 1

permission: active

accounts:

- permission:

actor: aliceacct

permission: active

weight: 1

---

Figure 12.



Note this model only needs 2 of the 3 conditions (bob’s key, alice’s account and the time-delay) to be met for the transaction to be considered valid (as symbolized by the +1/2 thresholds in **Figure 13**)

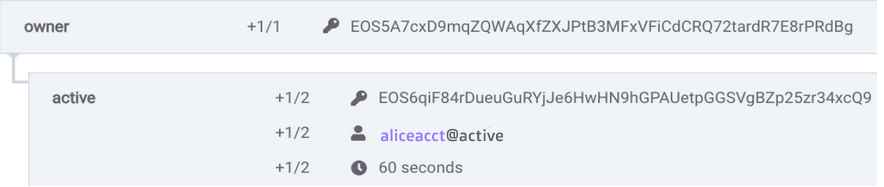


Figure 13.



**Very Strong Security:** nested linkauth with dedicated keys

So far we have seen account / key security, but we can have further granular security by making certain smart contracts accessible and others not. To do this we use **linkauth**.

**Definition**

**linkauth** assigns a **specific action** in a smart contract to a dedicated permission you have created. This allows you to assign a key or an account to **only** have access to **predefined** actions, limiting their ability to cause damage to your account should they become compromised.

**Permission Creation**

Let’s say, Bob wants to run a lottery smart contract with an action **claimrewards**. Not everyone should be able to call the cashrewards action. To limit this action to a specific permission, i.e. the winner, the first step is to create a new permission crafted in a .yaml file as exemplified by **Figure 14.**

---

threshold: 1

keys:

- key: EOS6hzSMaTPk6bKpADiAHUS4sYcWWjHbPof6FkQSQ4LgWh9SaWMLi

weight: 1

permission: claimer

---

Figure 14. **lottery**.yaml

This must then be updated to a new **claimer** permission*.* To do this, Bob would need to execute the following command for a prospective **winneracct**:

**$ eosc system updateauth winneracct claimer owner lottery.yaml**

Bob can then assign the newly created claimer permission, to the **winneracct** who has the private key corresponding to the public key (**EOS6hzSMaTPk6bKpADiAHUS4sYcWWjHbPof6Fk QSQ4LgWh9S aWMLi**) in their possession with the command:

**$  eosc system linkauth winneracct claimrewards claimer**





Figure 15.

The updated claimer permission structure in **Figure 15** means that only the account that contains the private key corresponding to  **EOS6hzSMaTPk6bKpADiAHUS4sYcWWjHbPof6F kQSQ4LgWh9SaWMLi** can successfully call the **claimrewards** action. Bob can therefore transfer the private key to any candidate that wins the lottery, so that they can claim the reward from his smart contract.

**Note:**

In this setup, the active permission, as the **rightful parent** of the **claimer** permission, can also successfully call the **claimrewards** action!

**Maximum Security:** linkauth with segregated sibling streams

To adhere to the **security principle of least privilege,** there is one final security pattern that can be implemented. This involves removing the nested vulnerability of the **active permission** to universally call all functions regardless of child permission set underneath it. To do this, we set up a segregated linkauth permission structure through a number of **sibling streams**. This results in each sibling being unable to call actions that are directly mapped to another sibling, and not to it, as shown in **Figure 16.**

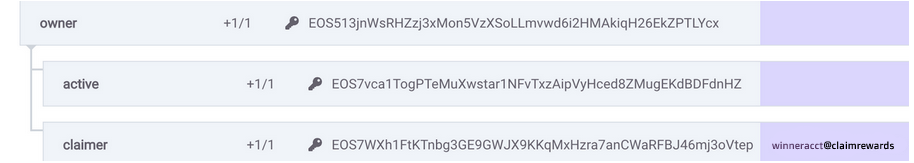


Figure 16.

Here the active account can call all actions **except** the **claimrewards** action which is **specifically** mapped to the **claimer** permission**.**

## About Klevoya

At Klevoya we know that building quality dApps is complicated. So we made it our mission to help EOSIO dApp developers deliver bug free, secure code; with our smart contract analysis and execution engine checking for bugs and security vulnerabilities is painless. Our team have deep blockchain expertise and have launched traditional software products that are used by millions.