A Foot in the Backdoor

Applying systems control theory to the 2024 xz utils backdoor cyberattack

Richard Bondi (rbondi@google.com)
John Thomas (jthomas4@mit.edu)

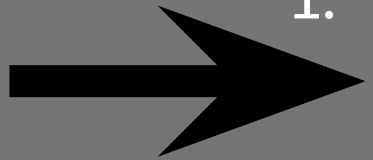
with Garrett Holthaus (gholthaus@google.com) & Ruben Barroso (rubenbarroso@google.com)
Presented on 2025-03-25 at the 2025 MIT STAMP Conference in Cambridge, MA (USA)

Outline

1. Purpose of the Analysis
2. CSD (Control Structure Diagram)
3. UCAs (Unsafe Control Actions)
4. Scenarios

5. Takeaways

Outline



1. Purpose of the Analysis
   a. What happened?
      i. Three concepts
      ii. The events
   b. Goals, Stakeholders, Losses, & Hazards
2. CSD (Control Structure Diagram)
3. UCAs (Unsafe Control Actions)
4. Scenarios

5. Takeaways

Supply chain attack

In supply chain attacks, malicious code or hardware parts enter before and during production, spreading via trusted suppliers undetected. Example: sabotaged rivets at a small factory end up in every U.S. Air Force plane, hiding in plain sight.

Root

Root is a computer permission level with absolute power. Whoever holds it has limitless control over the machine.

Backdoor

A backdoor is a hidden passage into a computer system. It allows silent entry, bypassing all security.

So obviously it's bad when a cyberattack allows someone to enter your computer through a backdoor and also gain root.

What (almost) happened in 2024?

Root.
On hundreds
of millions
of computers.

Not a virus that spreads.

It shipped inside the operating system.

A backdoor inside a tiny 3rd party component.

In other words, a supply chain attack.

The component was xz utils.

A tiny file compression program.

The operating system was Linux.

Just before Linux's release,
a programmer noticed
an odd 0.5 second slowdown,
and uncovered the backdoor.

He reported it
on a security mailing list,
and the internet erupted.
It was caught just before
being widely distributed.

Okay:

You can breathe now.

And I'll turn off the music.

Music by HorrorScapes from Pixabay.

Let's talk about Linux.

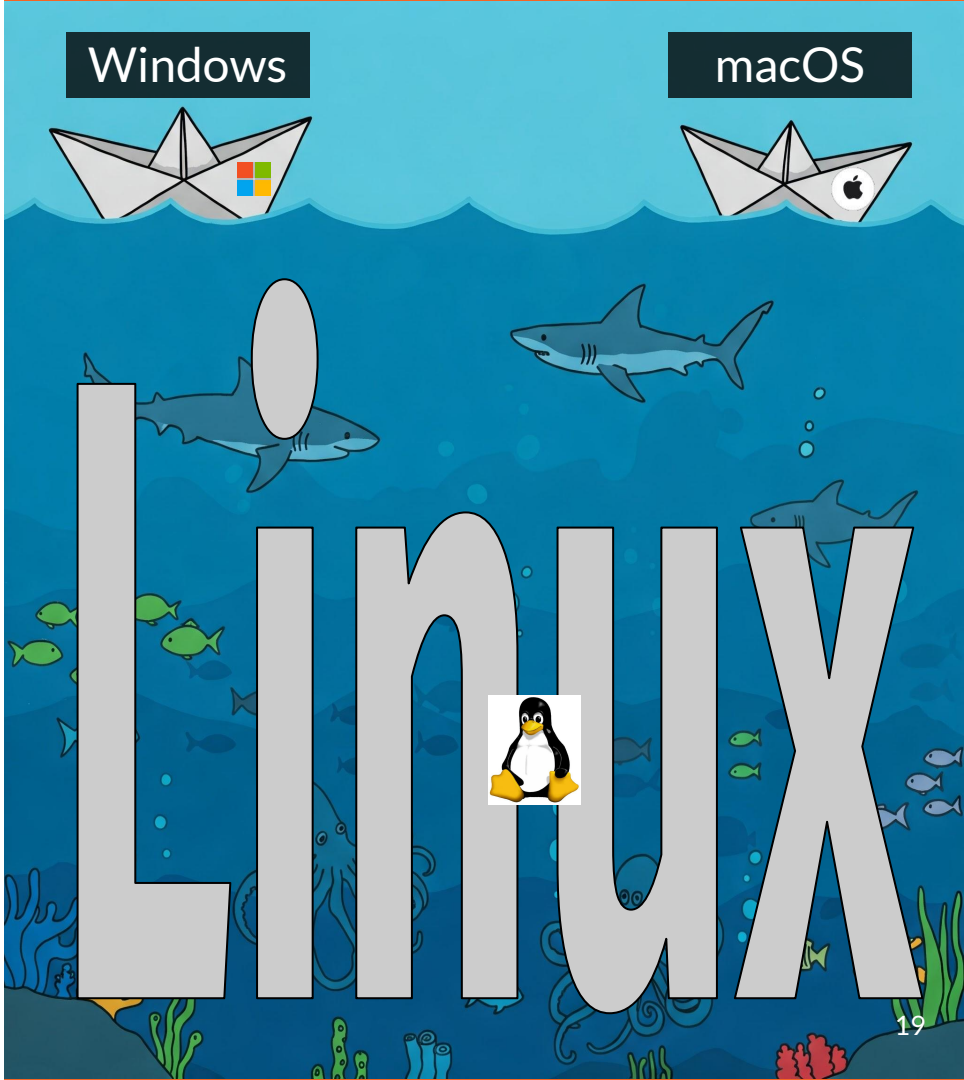
Linux is an operating system, like Microsoft Windows or Apple's macOS.

Here are three things to know about it.

70% of earth is water.

70% of the Internet is Linux.

Your Windows or macOS laptop is one of under 30% of all computers. Most online services—banks, Amazon, stores—run on Linux servers, making desktop and laptop systems mere paper boats on a vast Linux ocean.





Closed source.

Yours to use, never to see.

Second, the source code for Windows and macOS belongs legally to Microsoft and Apple.
To see it, you'd have to steal it, risking jail and fines.
Linus ix open source: anybody can download the source code, and change it.

And they do.



Open source.

Download it free.



Third, there are hundreds of flavors of Linux.

Each is maintained by its own team of volunteers.

Over the decades, each team has evolved its own processes and quality control.

We analyzed:

In this cyberattack, the attacker slipped the backdoored version of xz utils into the handful of the most widely used Linuxes.

We analyzed how he did it for one called Debian.

It's one of the most popular.

Debian

In this talk, you'll here "package" a lot.

On Linux, a "package" is an app installer

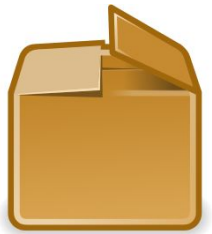
Microsoft Windows



Apple macOS



Linux



Outline

1. Purpose of the Analysis
    a. What happened?
    b. Goals, Stakeholders, Losses, & Hazards
2. CSD (Control Structure Diagram)
3. UCAs (Unsafe Control Actions)
4. Scenarios

5. Takeaways

STPA-ish

Let's start as if this were an STPA of supply chain vulnerabilities of Linux, with goals, losses, etc.

STPA step 1: *Work in progress!*

Linux runs virtually everywhere, so the stakeholders are practically anyone in a modern society!

Stakeholders

By sector:

S1. Public services & Government

S2. Private sector & industry

S3. Critical infrastructure providers

S4. Open Source Ecosystem

S5. General users

Any payload could be put through the backdoor: ransomware, keystroke loggers, file stealers, anything. So the losses are very broad, and in the worst case enormous.

Losses

Loss of:

L1: Life

L2: Critical protected information

L3: Business

L4: Mission

L5: Trust/Public confidence

But there are only five hazard states

Hazards

H1: A **package** is **bad**

H2: Linux development process is unable to block **bad package**

H3: Linux contains **bad package**

H4: Linux processes are unable to remove a **bad package** from a Linux release

H5: Linux processes are unable to remove a **bad package** from users' already installed Linux (i.e. via mailing list warnings).

A **bad package** is code that is, malicious (e.g. has a backdoor), privacy-violating, unstable, untested, security-vulnerable, or out-of-date.



And as Gary Oldman might yell:

Every hazard has all the losses!

H -> L

H1	L1, L2, L3, L4, L5
H2	
H3	
H4	
H5	

Question

Can we anticipate security flaws that today's methods miss?

Answer

To answer, we had to learn **what** flaws were overlooked.

And **how**.

Purpose of the analysis:

We wanted to prevent those losses.

Prevent the losses.

But we also wanted to see if our method could do that better than existing methods, and...

But better.

And live.

... get actual programmers to start using it in the field.

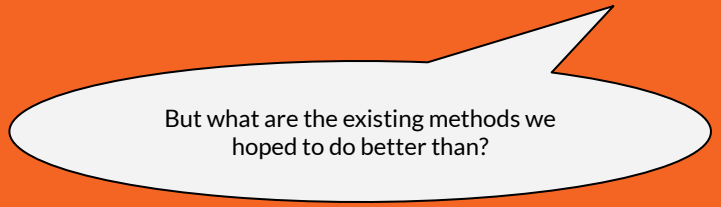
Purpose of ~~the analysis~~ this talk:

So the point of this talk is our control loop method — a building block of STPA and CAST — rather than the discovered hazards and mitigations.

A simple method anyone can use.

That means *you.*

Better than what?



Safety engineering *has* **Hazard Analyses:**

STPA, Fault Tree Analysis, Probabilistic Risk Assessment, Human Factors Analysis, etc.

Security engineering *has* **Threat Modeling:**

Next slide...

STRIDE

Microsoft

PASTA

Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis | Wiley

THE HYBRID METHOD

The Hybrid Threat Modeling Method:

SQUARE
Security Cards
Personae Non Gratae

ATTACK TREES

Schneier et al.

Incomplete list... and none needed for this talk :-)

THREAT MODELING MANIFESTO

Threat Modeling Manifesto

Diamond Method

The Diamond Model of Intrusion Analysis

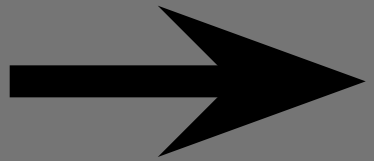
CYBER KILL CHAIN

Cyber Kill Chain® | Lockheed Martin

MITRE ATT&CK

https://attack.mitre.org/

Outline



1. Purpose of Analysis
2. CSD (Control Structure Diagram)
3. UCAs (Unsafe Control Actions)
4. Scenarios

5. Takeaways

Our next step is the control structure diagram.

Here it is!
We can cut through its complexity by focussing on a few boxes.

STAMP Control Structure Diagram of how the May 2024 xz utils backdoor got into Debian

Control Structure Diagram of 2024 xz utils backdoor infiltration into Debian © 2025 by John Thomas, Richard Bondi is licensed under CC BY-ND 4.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-nd/4.0/.

On the Internet, **Debian** is one of many popular and freely available variants of **Linux**, called **distros** (for "distributions"), each maintained by its own community of volunteers. The volunteers modify Linux, and curate many optional 3rd party programs that enhance or are critical to it. These 3rd party programs and how they get into a distro is sometimes called a **supply chain**. For example, the xz utils program is written by somebody who isn't a Debian volunteer, but is then curated and put into a so-called Debian **package** by Debian volunteers in order to ship with Debian.

In 2024, an unknown attacker succeeded in getting a modified version of xz utils into many Linux distros that would have given him or her secret and complete access to hundreds of millions of computers running Linux. This is a control structure diagram of how they infiltrated it into Debian, and how it was caught.

Legend

Source Code developers / managers / owners | Communities | Code management machinery | Source code & Debian package pipeline

Term	Meaning
OSS	Open Source Software
SP	Sock Puppet
Debian	A free and open source Linux distribution
DPM	Debian Package Maintainer
DD	Debian Developer
NMU	Non-maintainer upload: When a package is uploaded to Debian by a DD instead of its DPM.
upstream	Software to be integrated into Debian
Governor	This diagram's abstraction of the Debian build machinery to allow the automated building and packaging of a project in Salsa.
Debian Approver	Someone who can approve an xz utils package upload: All DDs, and the xz utils DPM.

Normal upload process for getting a new xz utils version into Debian:

1. **DPM copies xz utils source** etc. to e.g. DPM's laptop
2. **DPM creates tarball** file from source etc.
3. **DPM puts tarball** on ftp-master.debian.net
4. **DPM effectively tells Governor**: "Allow Autobuilder to build xz utils."
5. **Automated copier builds xz utils source** into a Debian package and puts it into **Unstable**.
6. "Radical" high risk tolerance users download and test it.
7. If they report no problems after ~2 days, it is copied automatically into **Testing** for medium risk tolerance users.

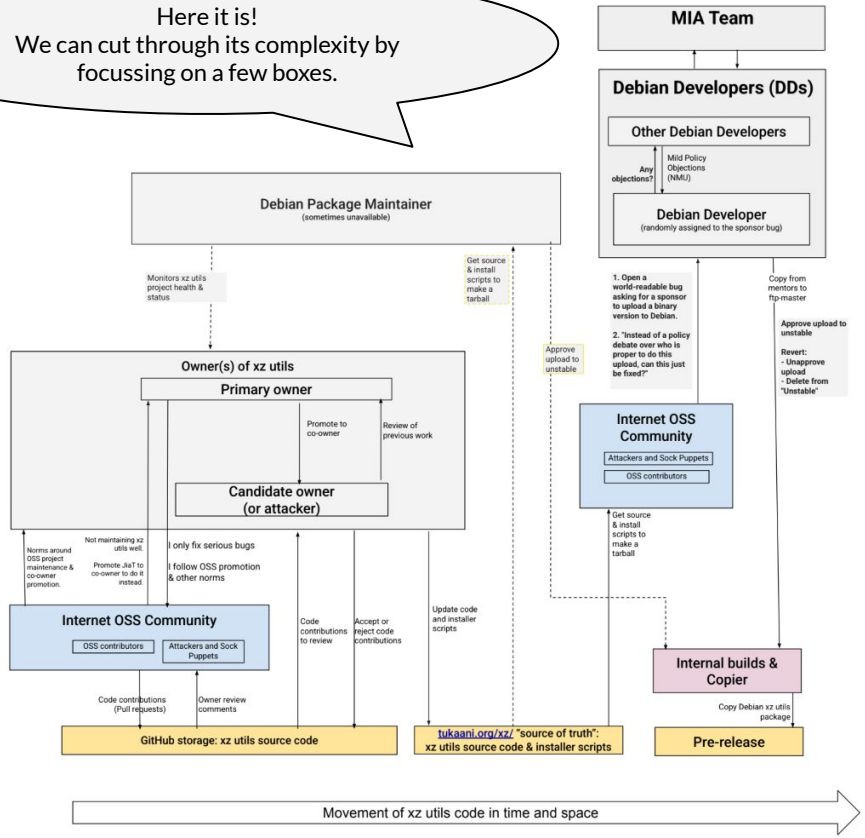
Normally happens, but didn't in this attack.

Downward control arrows represent control actions, responsibilities, enforcement of constraints, and updates to process models within the control structure.

Upward feedback arrows represent process feedback, execution status, sensor data, error reports, model updates, and safety constraint violations, all of which inform controllers about the state of the system.

Actual upload process followed for the backdoor attack:

1. **SP1 presumably copied** from xz utils source to e.g. SP1's laptop.
2. **SP1 created tarball** file from source etc.
3. **SP1 put tarball** on mentors.debian.net
4. **SP1 opened Debian bug requesting sponsorship,** asking a DD to upload xz utils from mentors.debian.net to Debian
5. **DD1 reassigned the bug** to the relevant package and retitled it as a request for a new upstream version of xz utils.
6. **SP2 and SP3 supported the update**, emphasizing its importance.
7. **DD2 criticized the use of a non-maintainer upload (NMU)** for a major update and asked where the DPM was who should upload
8. **DD3 responded** stating he had been acting as the de facto DPM was considering officially taking over from its official DPM.
9. **SP2 bullied the DDs, urging prioritizing the fix** over policy debates.
10. **DD3 copied xz utils** from mentors.debian.net to ftp-master.debian.net
11. **DD3 effectively told the Governor:** "Allow automatic builder to build xz utils."
12. **Automated copier builds the xz utils source** into a Debian package puts it into **Unstable**.
13. **Andres Freund reported that** the **Unstable** xz utils package has a backdoor. This happens before it is automatically copied into **Testing**.
14. **DDs removed it** from **Unstable** and **ftp-master.debian.net**.

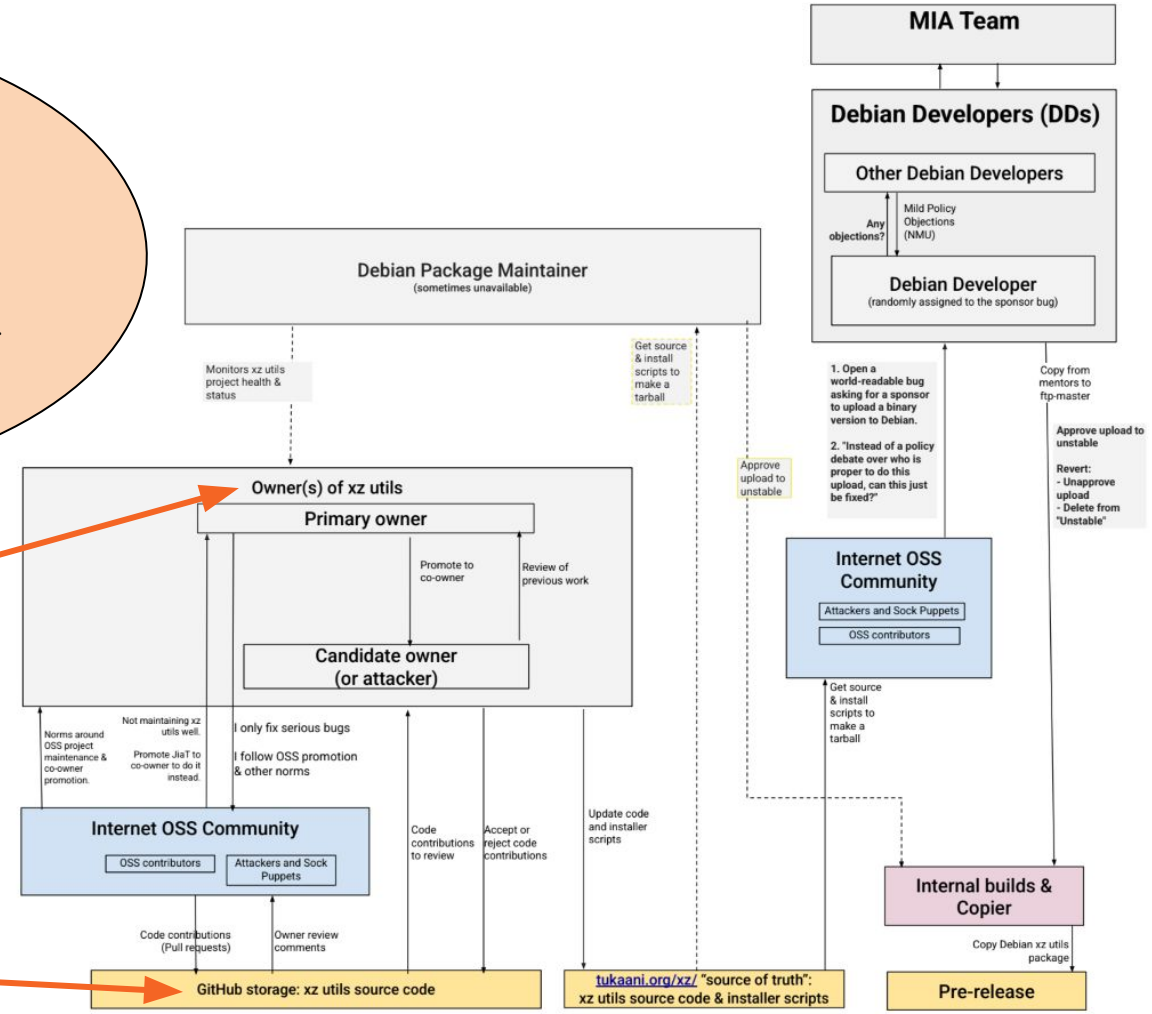


In open source software, a software packages is often written and maintained by just a single Owner.

The code lives on websites like GitHub, where anyone can send the owner code changes and bug fixes to accept or reject. That's the owners responsibility, as well as to announce new versions. Linux's like Debian may add these versions to their new Linux releases.

Owner

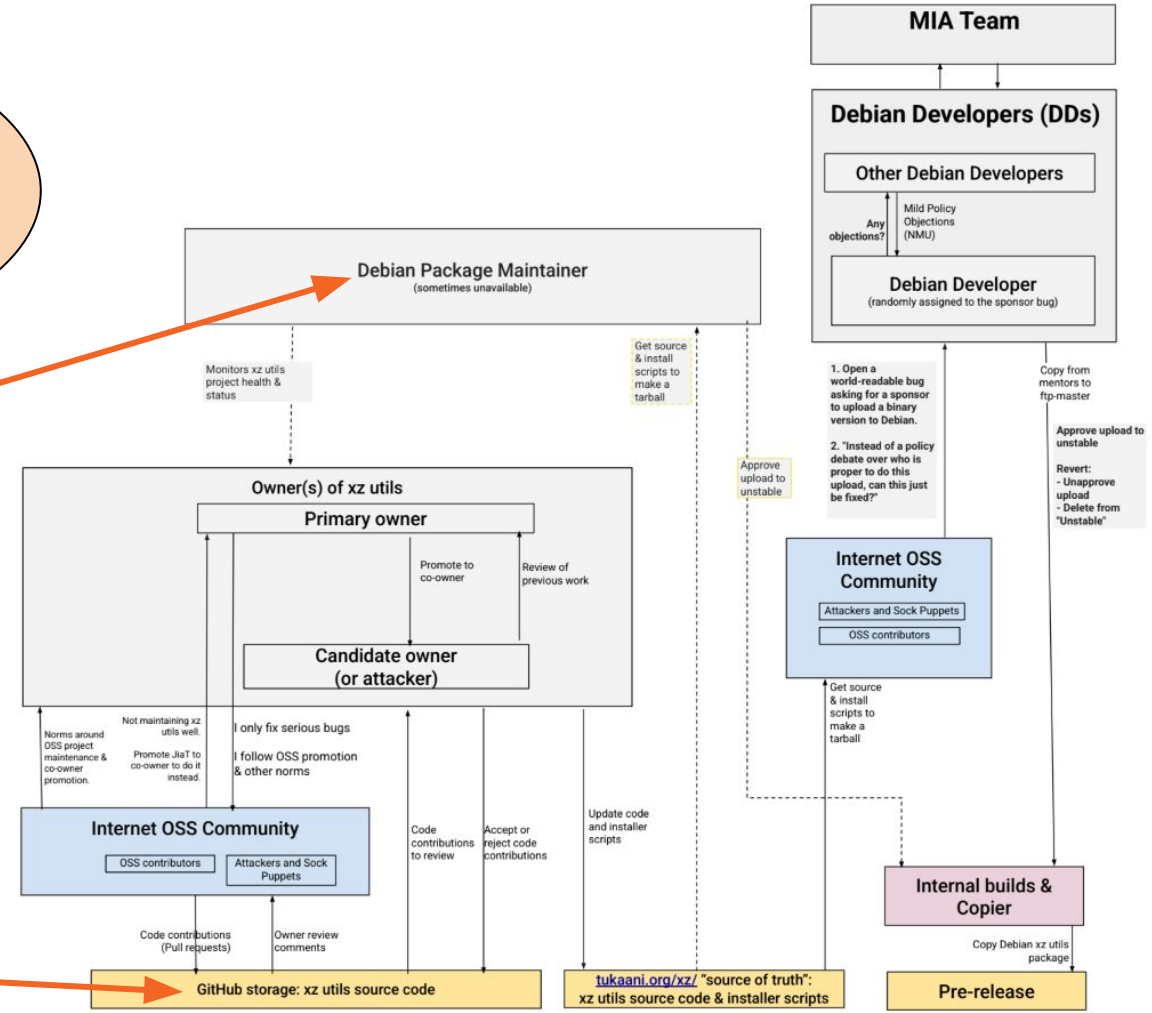
GitHub.com



A Debian Package Maintainer is a Debian volunteer whose responsibility is to know the owner, know the code, decide whether its ready to upload to Debian, and then actually upload it.

Maintainer

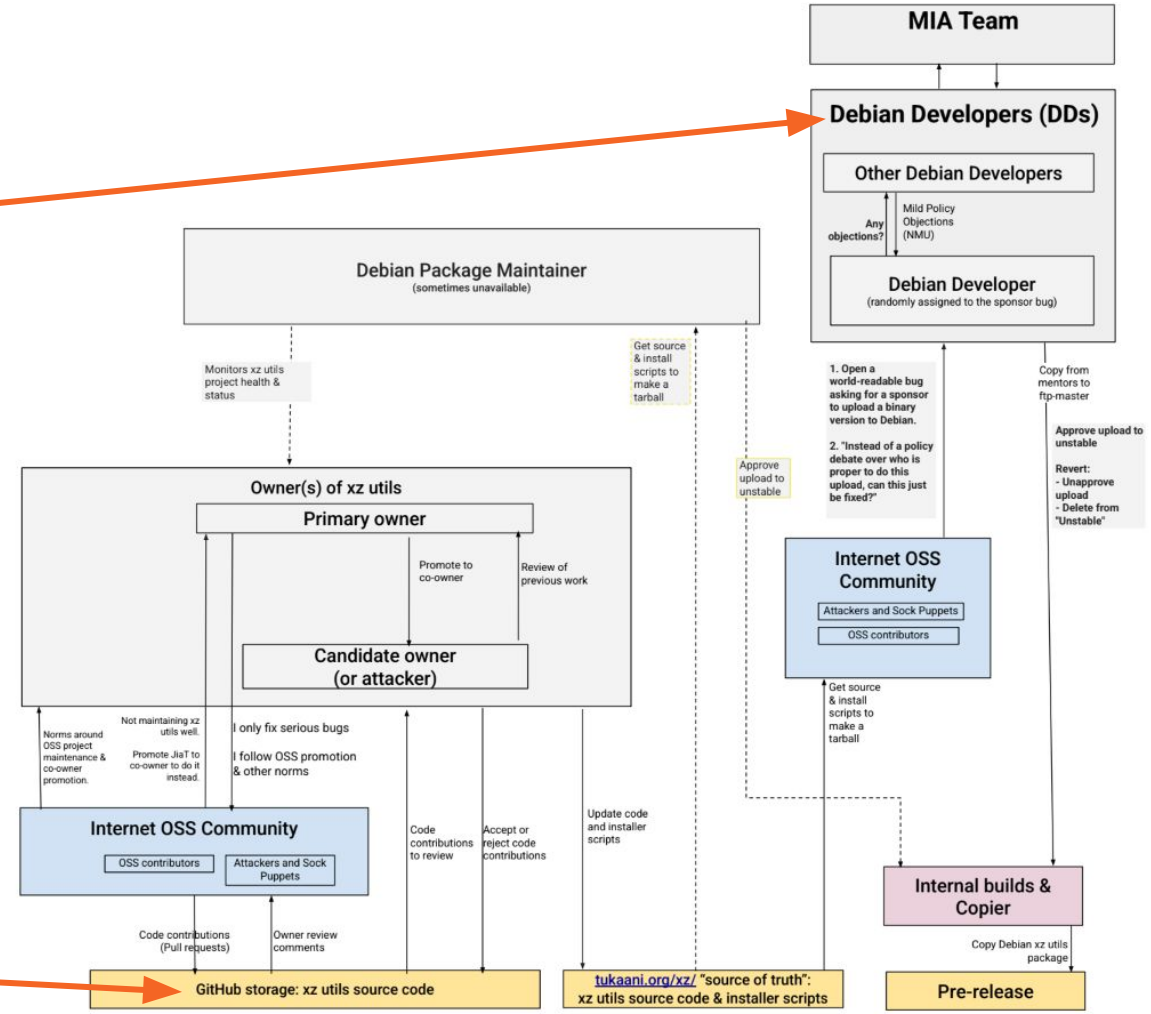
GitHub.com



Developer

A Debian Developer is the most senior position in Debian: they can upload anything and do anything.

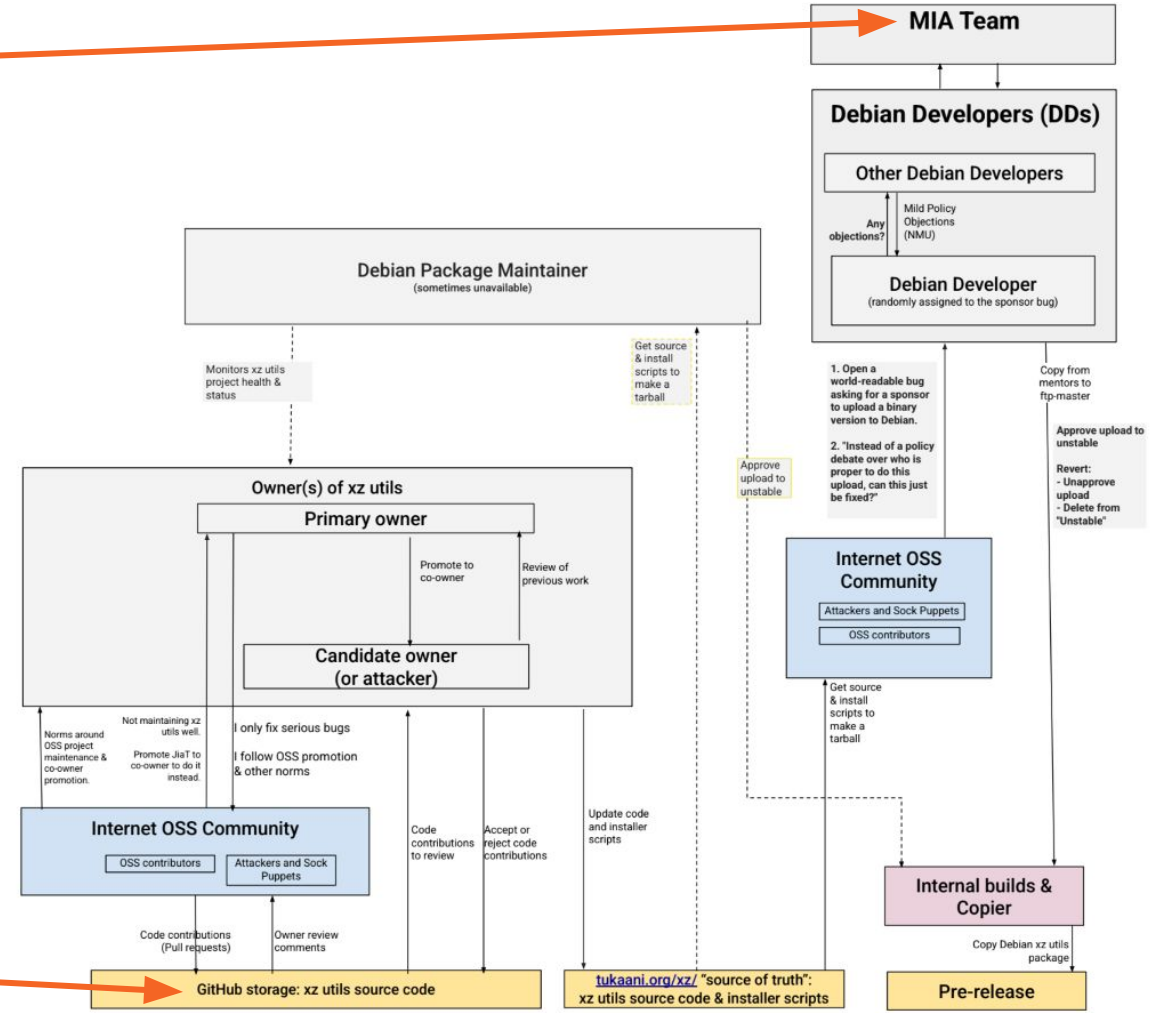
If the Package Maintainer is absent, they can upload the Maintainer's package — but this is seriously frowned upon and rare, because only the Maintainer knows the package well. That's called a Non-Maintainer Upload, or NMU.



GitHub.com

MIA Team

Last is the Debian Missing in Action, or MIA team. They are responsible for noticing when Package Maintainers disappear, to do wellness checks on them, and to help recruit a replacement. There's a whole MIA process for this.



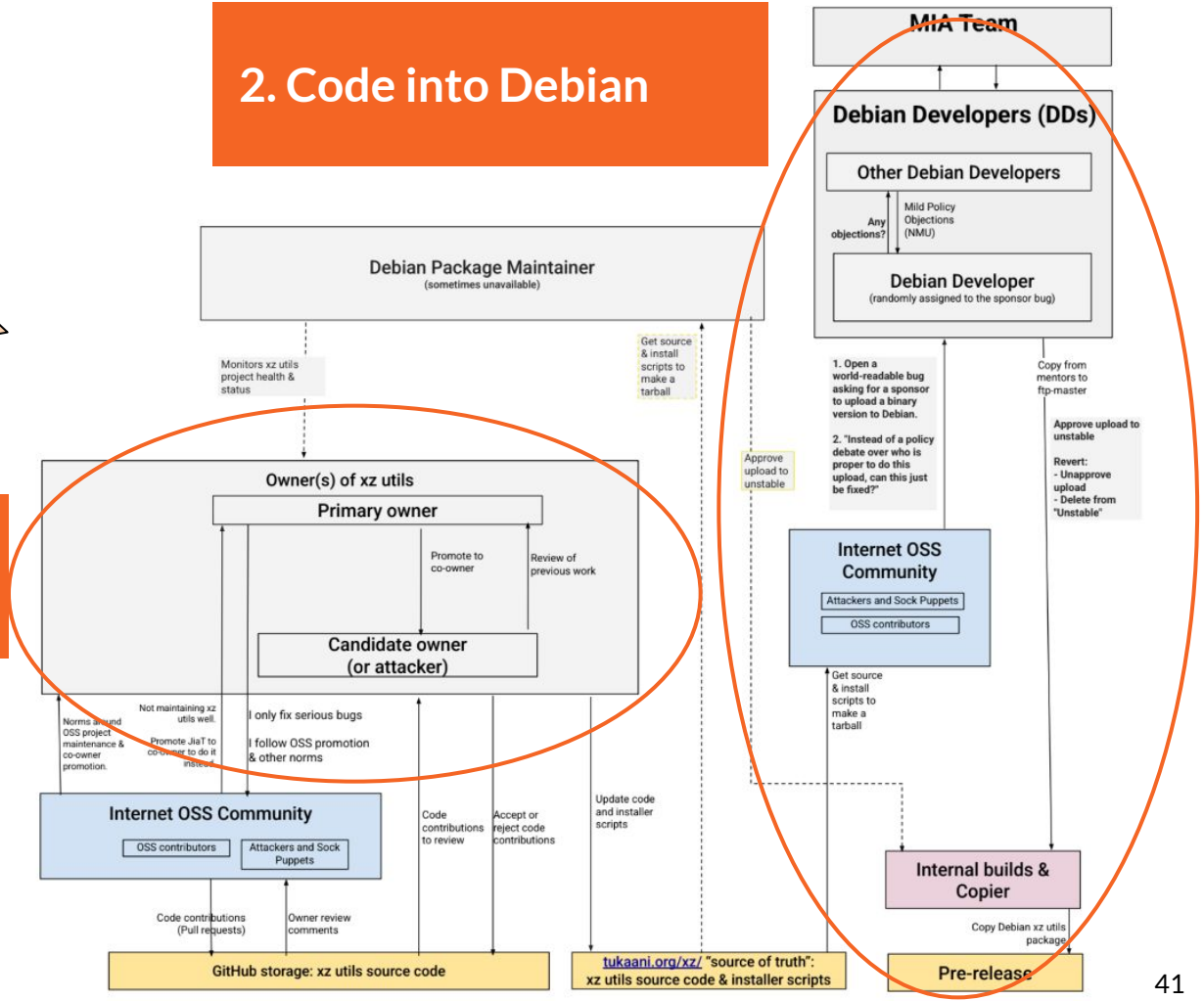
GitHub.com

The main attack events were:

1. The attacker got the owner to make him a co-owner, and then secretly created the backdoor .
2. When the package maintainer was absent, the attacker got Debian to accept the backdoored xz utils.

1. Become co-owner

2. Code into Debian



How'd you do that?!

This attack was so complicated, it took three years! Three years!

So how does anyone diagram that?

I'll show you.

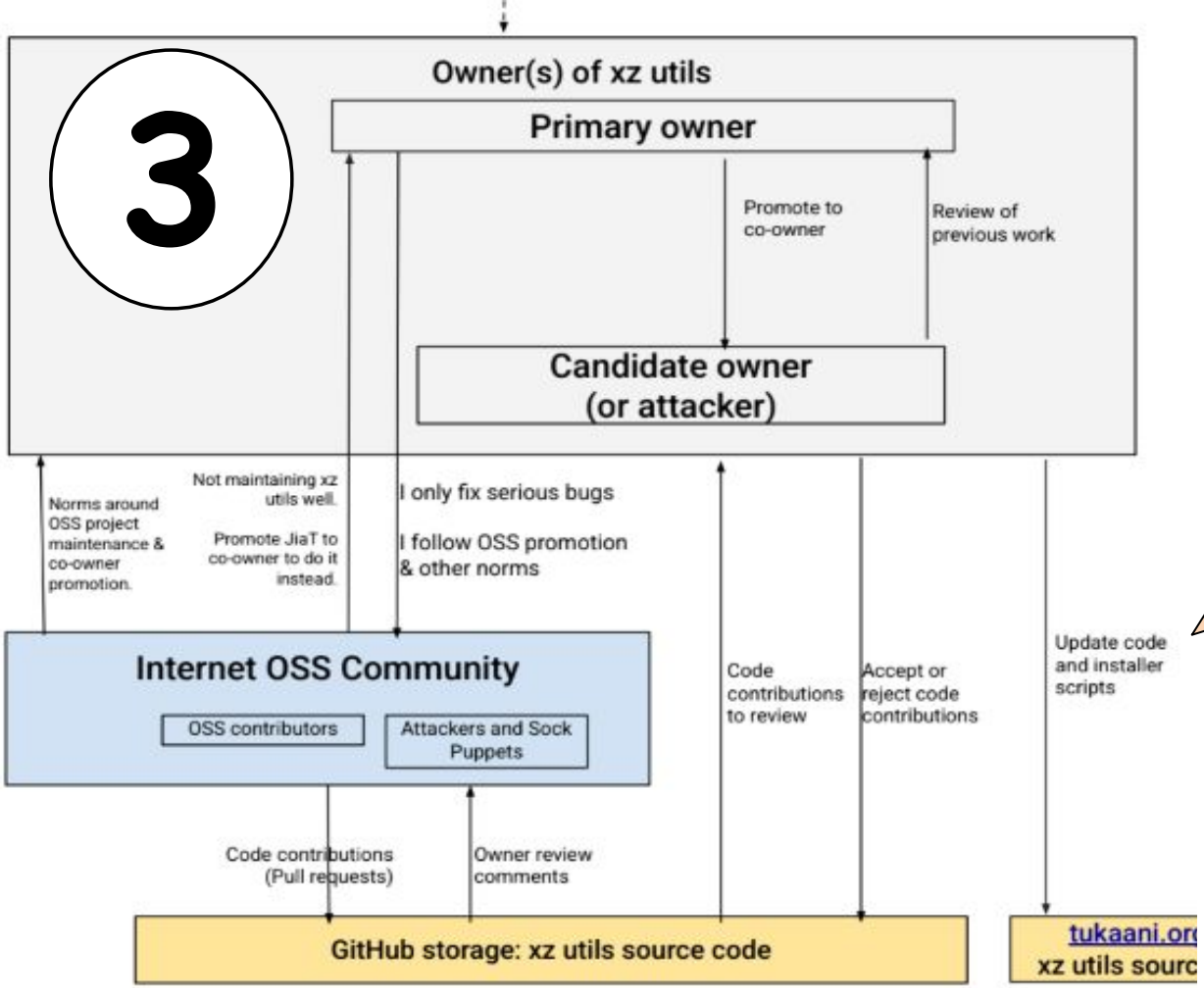


I was an SME on this project, and this is what it felt like.

First, I'd tell the STPA facilitators a long complicated attack story.

Second, invisible magic happened.

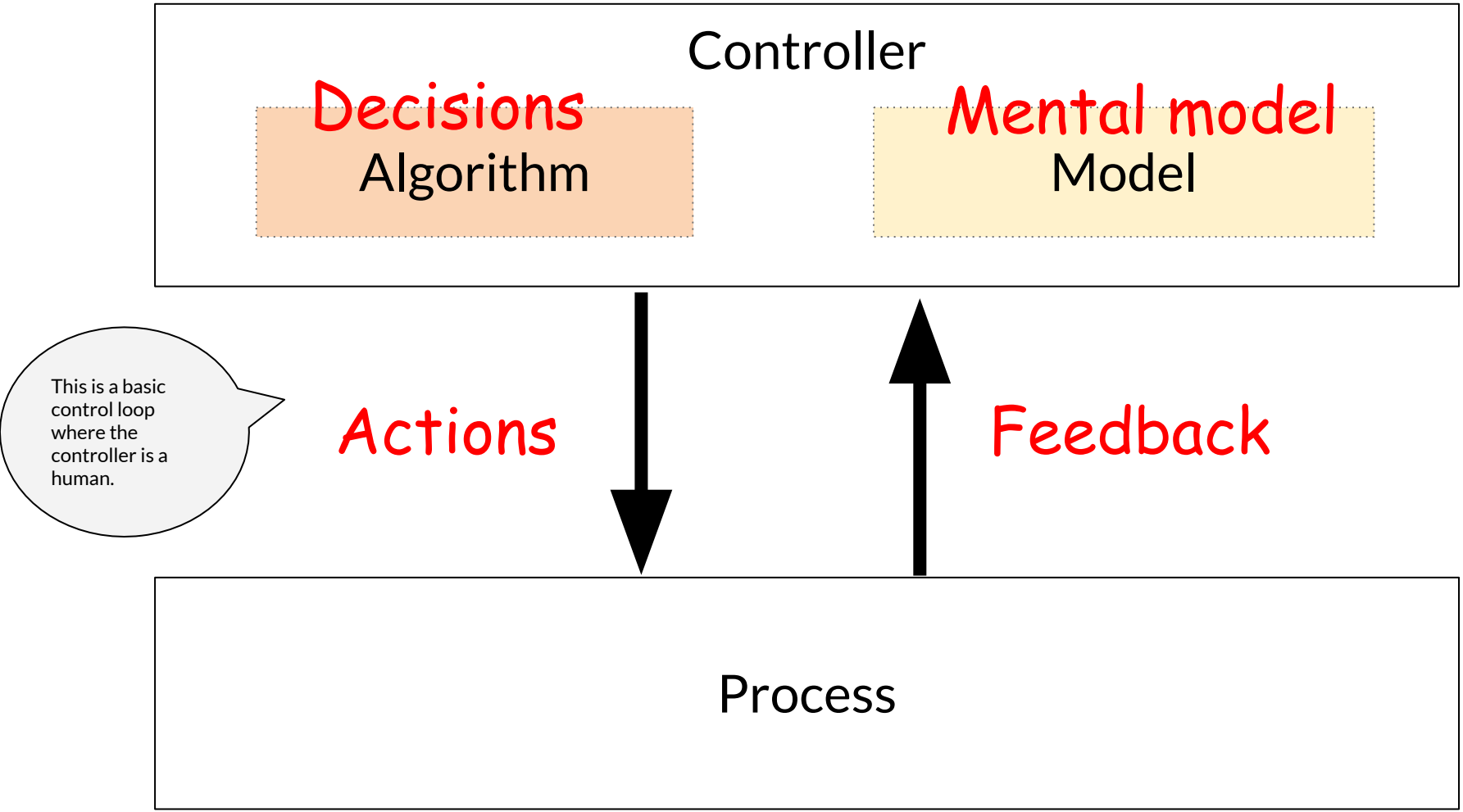




Third, a clear & compelling control loop appeared.

One day, they showed me the magic.

It's a trick.


Controller
Decisions
Algorithm
Mental model
Model
This is a basic control loop where the controller is a human.
Actions
Feedback
Process

Lasse Actions

Actions

- only working on bugs at this time
- leaning towards making Jia co-maintainer
- set high bar of qualifications for co-maintainer
- lists Jia as member of tikaani
- sets bug report email address to group email, now goes to Jia

Jia Tan becomes co-owner

Feedback to Lasse

Feedback

- time pressure: why isn't patch done?
- MIA: owner doesn't have time
- suggestion: make jia a co-owner
- time pressure: bullying
- time pressure: Jigar bullying
- time pressure: Dennis bullying
- suggestion: Jigar asks Jia why he can't commit patch himself
- acting co-maintainer: Jia posts full list of features in new version

This is an actual slide of the facilitator's notes while I was talking through one attack phase.

Whenever I said something that was an action, or feedback, or a decision, or described somebody's beliefs, they wrote it down as one of those.

Then they drew one controller and one process, and just filled it in.

That's the trick.

We built the diagram one loop at a time like this, combining them at the end.

Now you can do it too.

Decision-making

Algorithm

- if unable to keep up with expected level of service then reduce qualifications bar for new co-maintainer
- if owner is MIA, need to find a solution (someone to take over)

Mental model

Mental (process) model

- Lasse believed the expected level of service was not being met
- Lasse believed that the co-maintainer qualifications are flexible and can be changed to meet the needs of the community

Problem

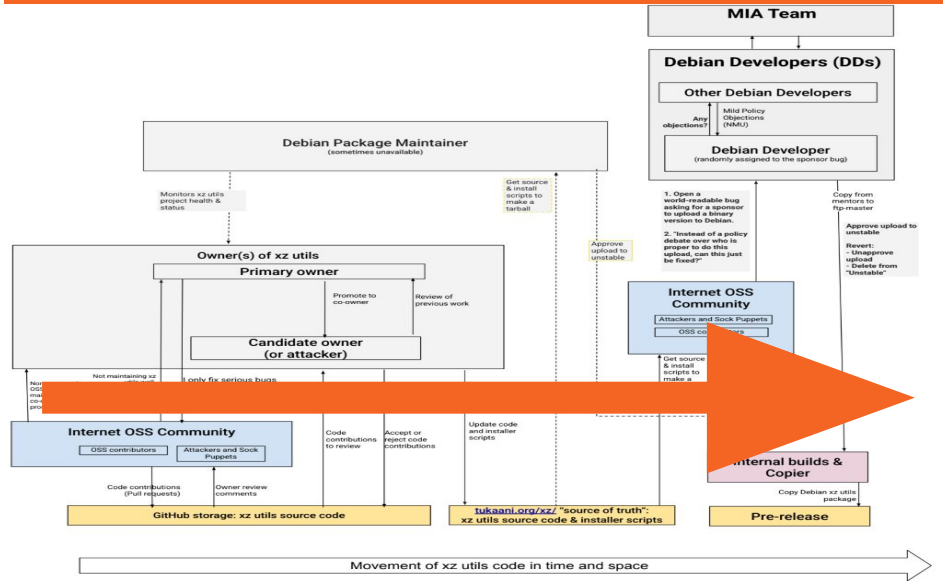
Many sequential loops

One final trick was how to show that the loops happened sequentially.

The solution: draw them from left to right, following the journey of the xz utils code in time and location.

Solution

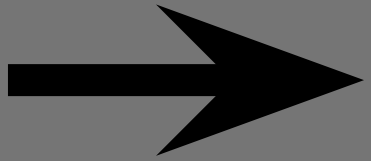
Left to right = time & space:



Outline

1. Purpose of Analysis
2. CSD (Control Structure Diagram)
3. UCAs (Unsafe Control Actions)
4. Scenarios

5. Takeaways

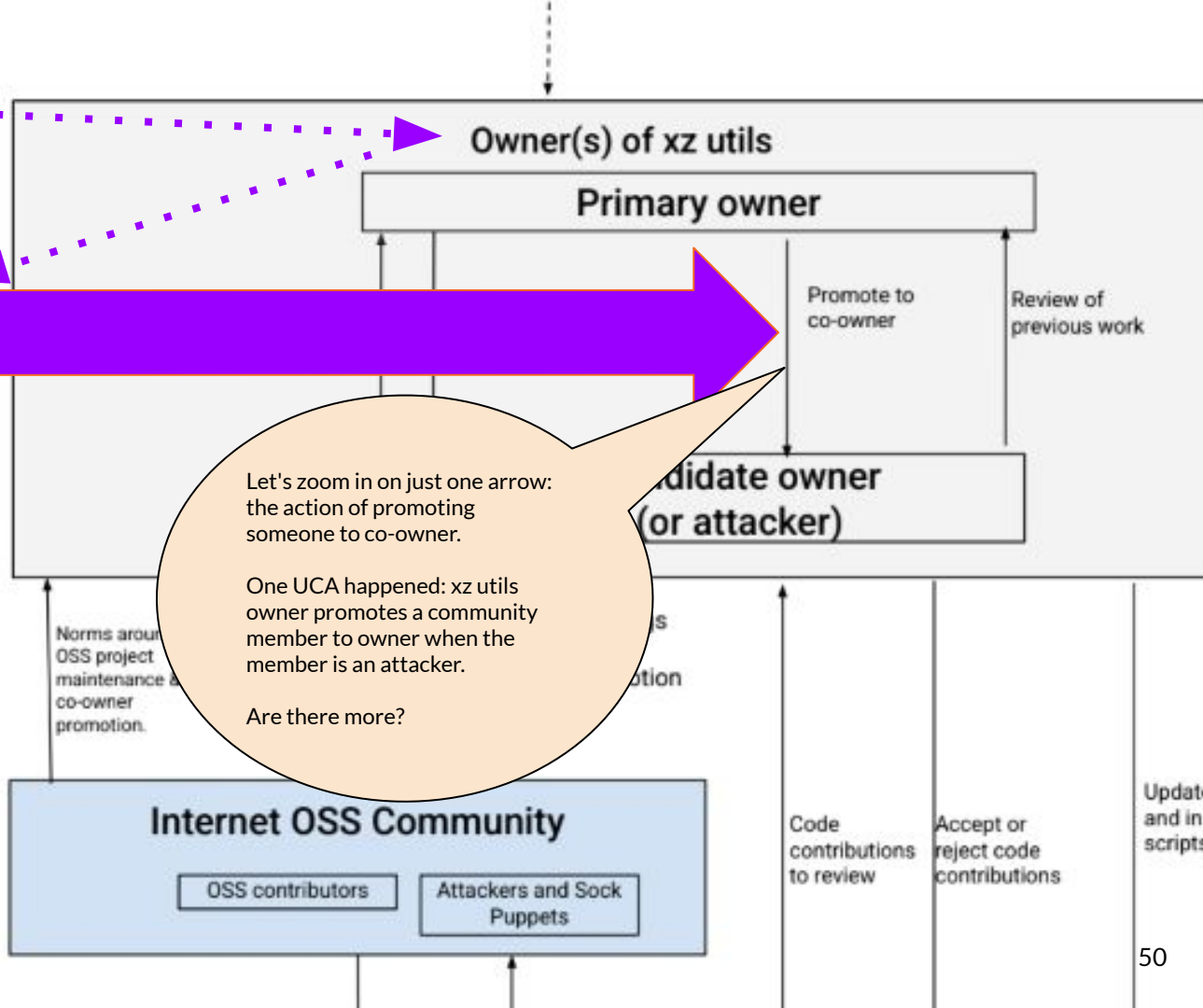


We've done the diagram.

Next up: UCAs.

Owner UCAs

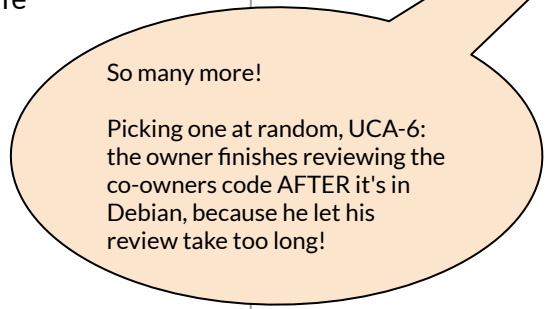
UCA: xz utils owner promotes a community member to owner when the member is an attacker



STPA Step 3: Unsafe Control Actions (UCAs)

For controller: xz utils owner

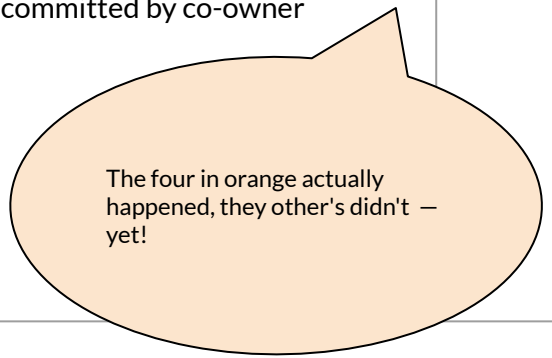
Not providing causes hazard	Providing causes hazard	Provided too late or too early causes hazard	Providing for too long, or not too soon, causes hazard
UCA-1: xz utils owner doesn't reject co-owner code when the co-owner commits malicious code UCA-2: xz utils owner doesn't approve code changes after it is committed by co-owner	UCA-3: xz utils owner promotes a community member to owner when the member is an attacker UCA-4 (provides): xz utils owner approves code from a stranger when it is unsafe	UCA-5: xz utils owner provides code rejection too late after the code is already introduced into Debian	UCA-6: xz utils owner continues prolonging review actions too long after the code gets integrated into Debian UCA-7: xz utils owner stops providing review actions too soon before code is finalized / accepted into Debian UCA-8: xz utils owner stops providing review actions before all package files have been scrutinized



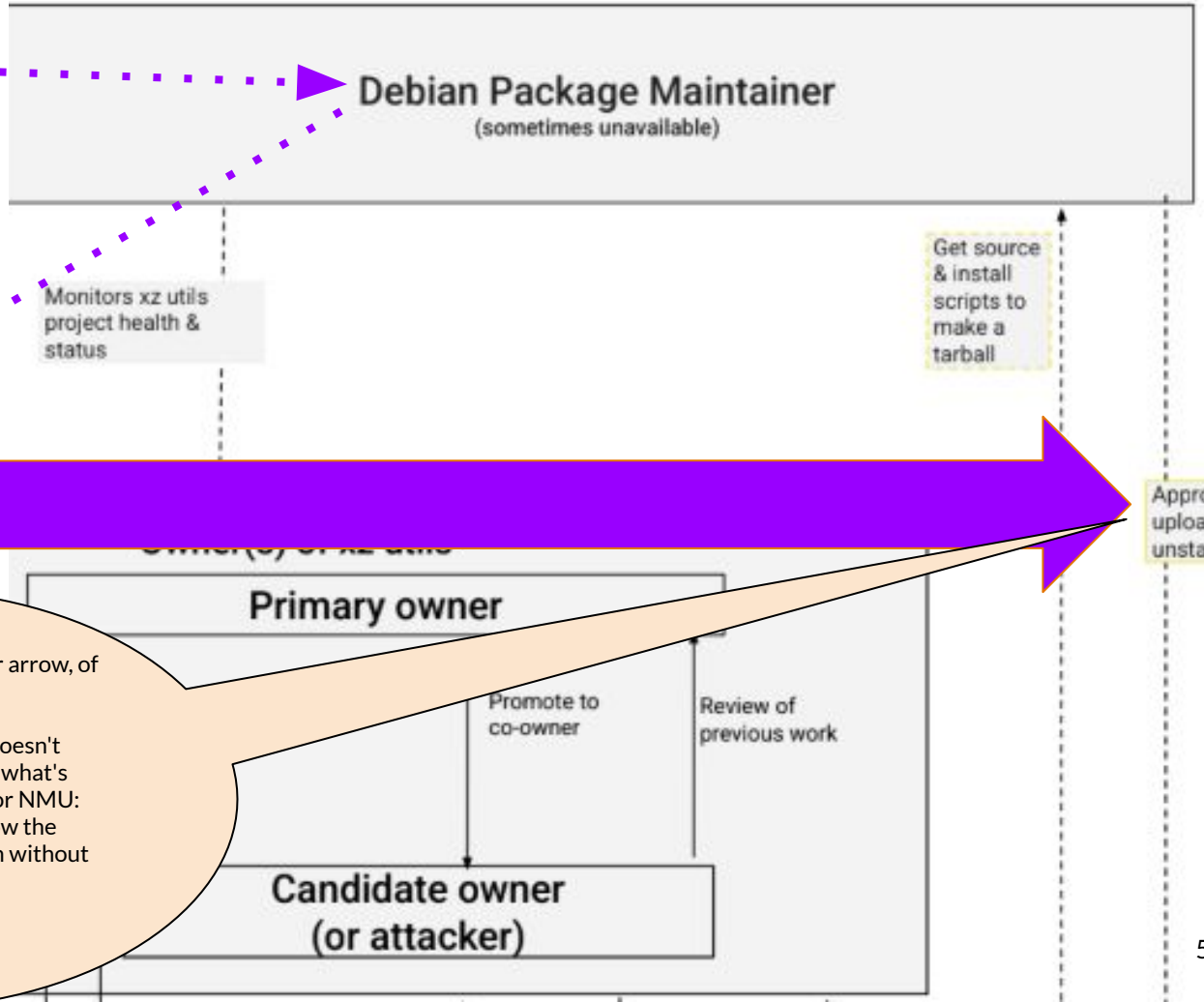
STPA Step 3: Unsafe Control Actions (UCAs)

For controller: xz utils owner

Not providing causes hazard	Providing causes hazard	Provided too late or too early causes hazard	Providing for too long, or not too soon, causes hazard
UCA-1: xz utils owner doesn't reject co-owner code when the co-owner commits malicious code	**UCA-3: xz utils owner promotes a community member to owner when the member is an attacker**	UCA-5: xz utils owner provides code rejection too late after the code is already introduced into Debian	UCA-6: xz utils owner continues prolonging review actions too long after the code gets integrated into Debian
UCA-2: xz utils owner doesn't approve code changes after it is committed by co-owner	UCA-4 (provides): xz utils owner approves code from a stranger when it is unsafe		**UCA-7: xz utils owner stops providing review actions too soon before code is finalized / accepted into Debian**
			UCA-8: xz utils owner stops providing review actions before all package files have been scrutinized



Maintainer UCAs



UCA: Maintainer doesn't review package when it is uploaded

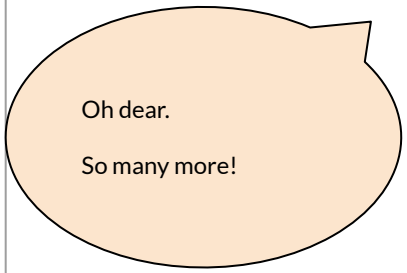
Now let's pick package maintainer arrow, of putting xz utils into Debian.

One UCA happened: maintainer doesn't upload package causing hazard of what's called a Non-Maintainer Upload, or NMU: Debian Developers who don't know the package well upload it into Debian without the Maintainer's review or input.

Are there more?

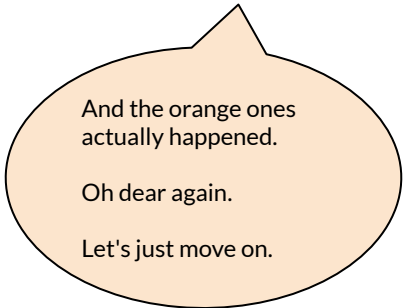
STPA Step 3: Unsafe Control Actions (UCAs)

For controller: Debian Package Maintainer

Not providing causes hazard	Providing causes hazard	Provided too late or too early causes hazard	Providing for too long, or not too soon, causes hazard
UCA-9: maintainer doesn't provide upload causing an NMU of a bad package UCA-10: maintainer doesn't provide review of an NMU of a bad package 	UCA-11: maintainer provides upload of a malicious package UCA-12: maintainer provides upload without monitoring owner and package project for suspicious events indicating an attack UCA-13: maintainer provides upload without monitoring package source code for suspicious changes UCA-14: maintainer provides upload without monitoring	UCA-16: maintainer does upload after package has already been NMUd UCA-17: maintainer does upload before a package is then NMUd	UCA-18: maintainer doesn't spend enough time reviewing package to detect malware before uploading it UCA-19: maintainer takes so long reviewing package that it causes an NMU

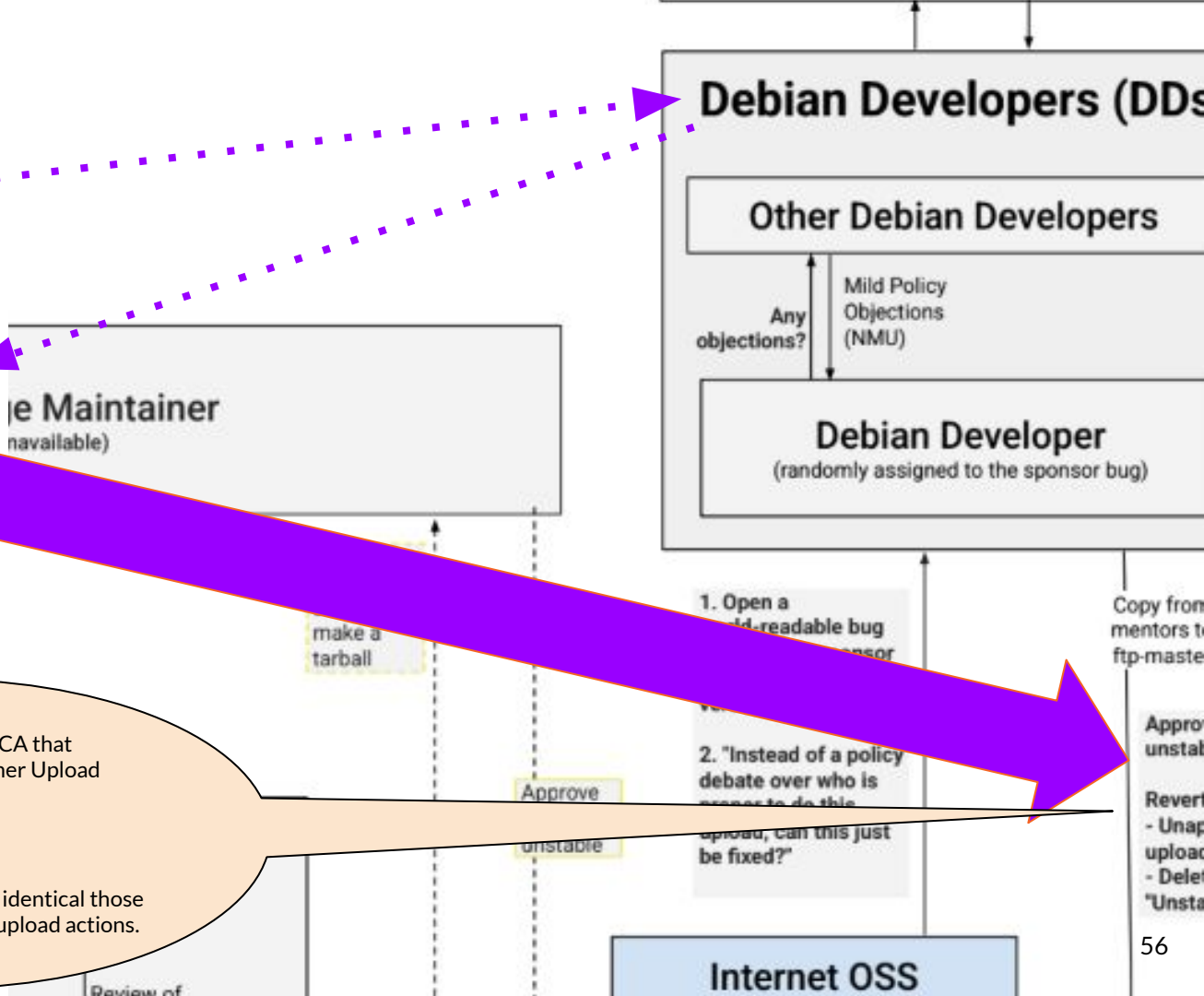
STPA Step 3: Unsafe Control Actions (UCAs)

For controller: Debian Package Maintainer

Not providing causes hazard	Providing causes hazard	Provided too late or too early causes hazard	Providing for too long, or not too soon, causes hazard
UCA-9: maintainer doesn't provide upload causing an NMU of a bad package	UCA-11: maintainer provides upload of a malicious package	UCA-16: maintainer does upload after package has already been NMUd	UCA-18: maintainer doesn't spend enough time reviewing package to detect malware before uploading it
UCA-10: maintainer doesn't provide review of an NMU of a bad package	UCA-12: maintainer provides upload without monitoring owner and package project for suspicious events indicating an attack	UCA-17: maintainer does upload before a package is then NMUd	UCA-19: maintainer takes so long reviewing package that it causes an NMU
	UCA-13: maintainer provides upload without monitoring package source code for suspicious changes		
	UCA-14: maintainer provides upload without monitoring		

Debian Developer UCAs

UCA: Debian developer provides upload without enough knowledge of the package



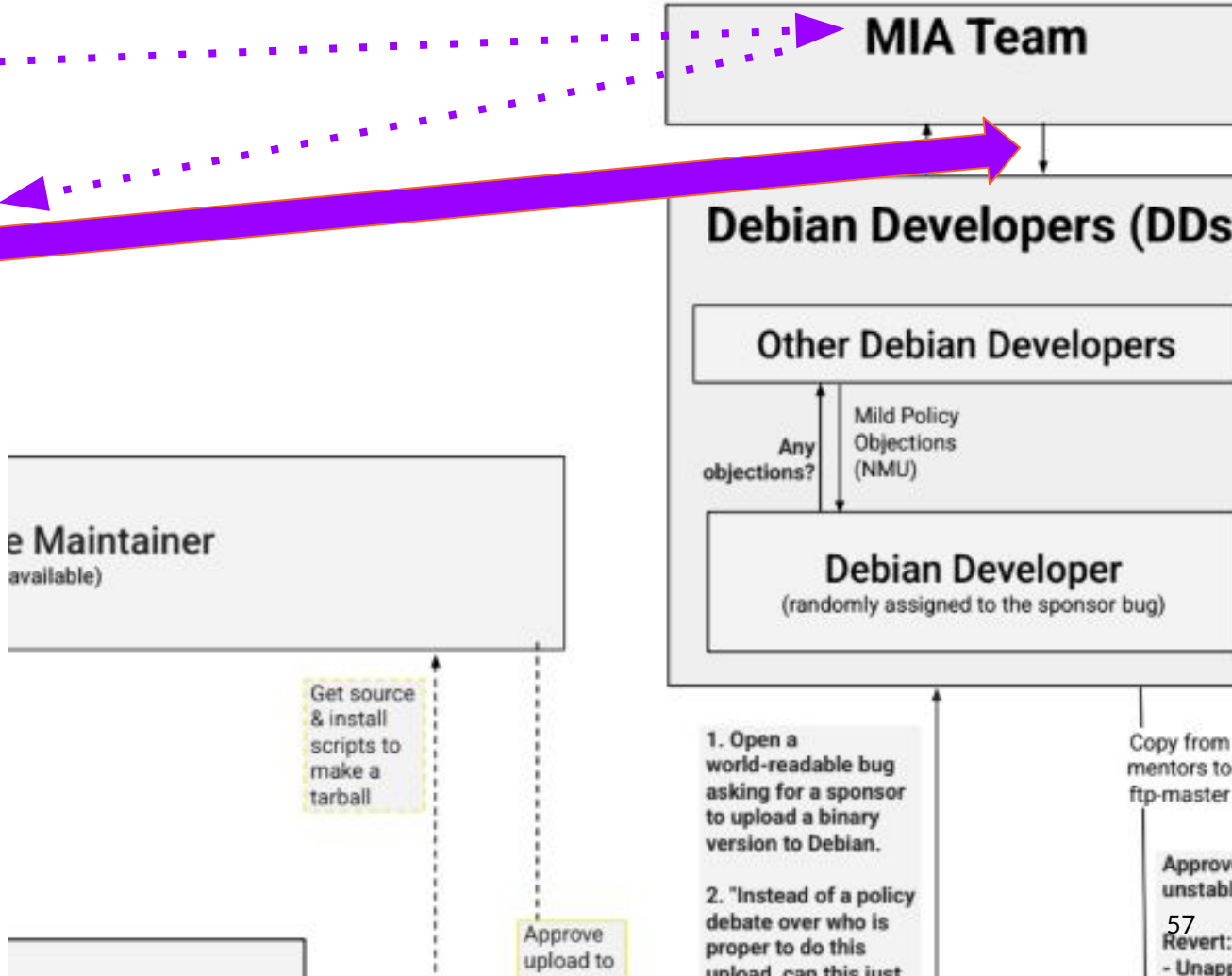
Here's a Debian Developer UCA that happened: that Non-Maintainer Upload (NMU) of xz utils.

Are there more?

Yes: the UCAs are essentially identical those of the Maintainer, i,e. unsafe upload actions.

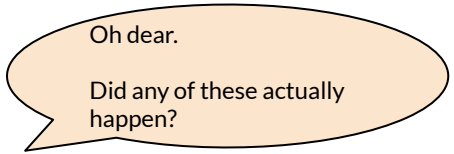
MIA Team UCAs

UCA: MIA team doesn't provide wellness check after 2 years



STPA Step 3: Unsafe Control Actions (UCAs)

For controller: MIA team

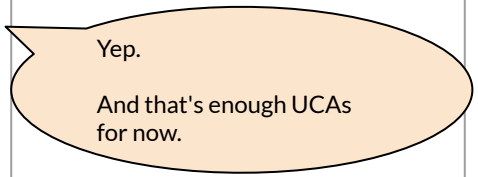


Not providing causes hazard	Providing causes hazard	Provided too late or too early causes hazard	Providing for too long, or not too soon, causes hazard
UCA-20: MIA team doesn't flag maintainer as MIA who is in fact MIA, causing NMU or Debian package to not get security updates UCA-21: MIA team doesn't flag package as orphaned that is in fact orphaned b/c maintainer is MIA, causing NMU or Debian package to not get security updates	UCA-22: MIA team flags package as orphaned that isn't orphaned, causing conflicting uploads by two maintainers UCA-23: MIA team flags package as not orphaned that is, causing NMU or Debian package to not get security updates	UCA-24: MIA team flags package as orphaned too late after wellness check, causing NMU or Debian package to not get security updates UCA-25: MIA team flags package as orphaned too early, before wellness check is complete, causing a maintainer upload and an NMU to conflict	UCA-26: MIA team takes too long to complete wellness check of MIA maintainer, causing NMU or Debian package to not get security updates UCA-27: MIA team takes too long after wellness check to flag package as orphaned, causing NMU or Debian package to not get security updates

STPA Step 3: Unsafe Control Actions (UCAs)

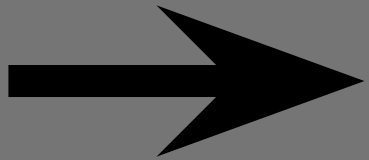
For controller: MIA team

Not providing causes hazard	Providing causes hazard	Provided too late or too early causes hazard	Providing for too long, or not too soon, causes hazard
UCA-20: MIA team doesn't flag maintainer as MIA who is in fact MIA, causing NMU or Debian package to not get security updates UCA-21: MIA team doesn't flag package as orphaned that is in fact orphaned b/c maintainer is MIA, causing NMU or Debian package to not get security updates	UCA-22: MIA team flags package as orphaned that isn't orphaned, causing conflicting uploads by two maintainers UCA-23: MIA team flags package as not orphaned that is, causing NMU or Debian package to not get security updates	UCA-24: MIA team flags package as orphaned too late after wellness check, causing NMU or Debian package to not get security updates UCA-25: MIA team flags package as orphaned too early, before wellness check is complete, causing a maintainer upload and an NMU to conflict	UCA-26: MIA team takes too long to complete wellness check of MIA maintainer, causing NMU or Debian package to not get security updates UCA-27: MIA team takes too long after wellness check to flag package as orphaned, causing NMU or Debian package to not get security updates



Outline

1. Purpose of Analysis
2. CSD (Control Structure Diagram)
3. UCAs (Unsafe Control Actions)
4. Scenarios (2 loops)
5. Takeaways



Now I'll show two ***scenarios*** that actually happened.

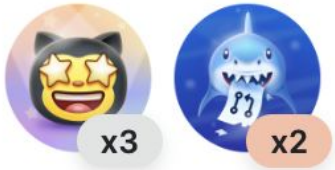
Jia Tan

JiaT75

Follow

781 followers · **1** following

Achievements



Block or Report

GitHub account of the attacker(s)

This is the public profile page of the attacker. Or attackers. Nobody actually knows except him. Or her. Or them. We've no idea. But almost certainly a nation state.

I'll call him "Jia Tan" and assume he was male.



Loop #1: Soft takeover of xz utils

The first loop happened outside of Debian.

It shows how Jia Tan created pressure on the owner to promote Jia Tan to co-owner sooner than he wanted to.

The loop is only about the pressure.

The actual promotion of Jia Tan is a separate loop we won't cover in this presentation.

Sock puppet

To understand what Jia Tan did, you have to know what a sock puppet is.

The photograph shows a sock puppet — put a sock over your hand, and give it eyes and hair.

In cyber security, a sock puppet is a fake email account in which you pretend to be someone else.

Jia Tan used lots of them.



Symbol

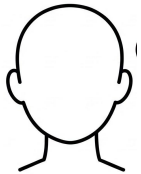


Actors

These are the actors in the control loop: the owner of xz utils, anyone on the internet, and Jia Tan and his sock puppets.



Owner of xz-utils



Anyone



Attacker (JiaT)

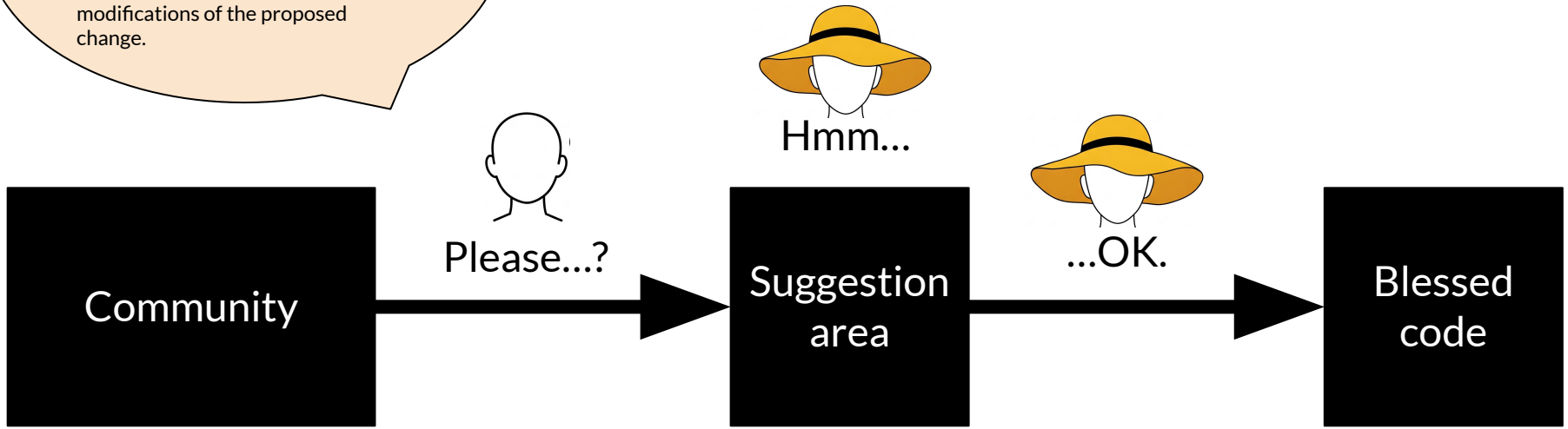


Sock puppet

What *should* happen

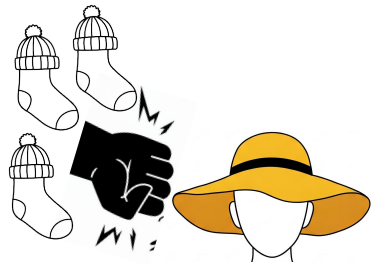
Anyone on the Internet can try to get a code change into xz utils.

Normally you submit it to a suggestion area, and the owner will either reject it or accept it. There is often a conversation that leads to modifications of the proposed change.



What happened

What actually happened here was that Jia Tan, after he had been winning the owner's trust for about six months, used sock puppets to suggest small changes, and when the owner didn't respond fast enough, had the sock puppets bully the owner, saying he should make Jia Tan a co-owner.





Community



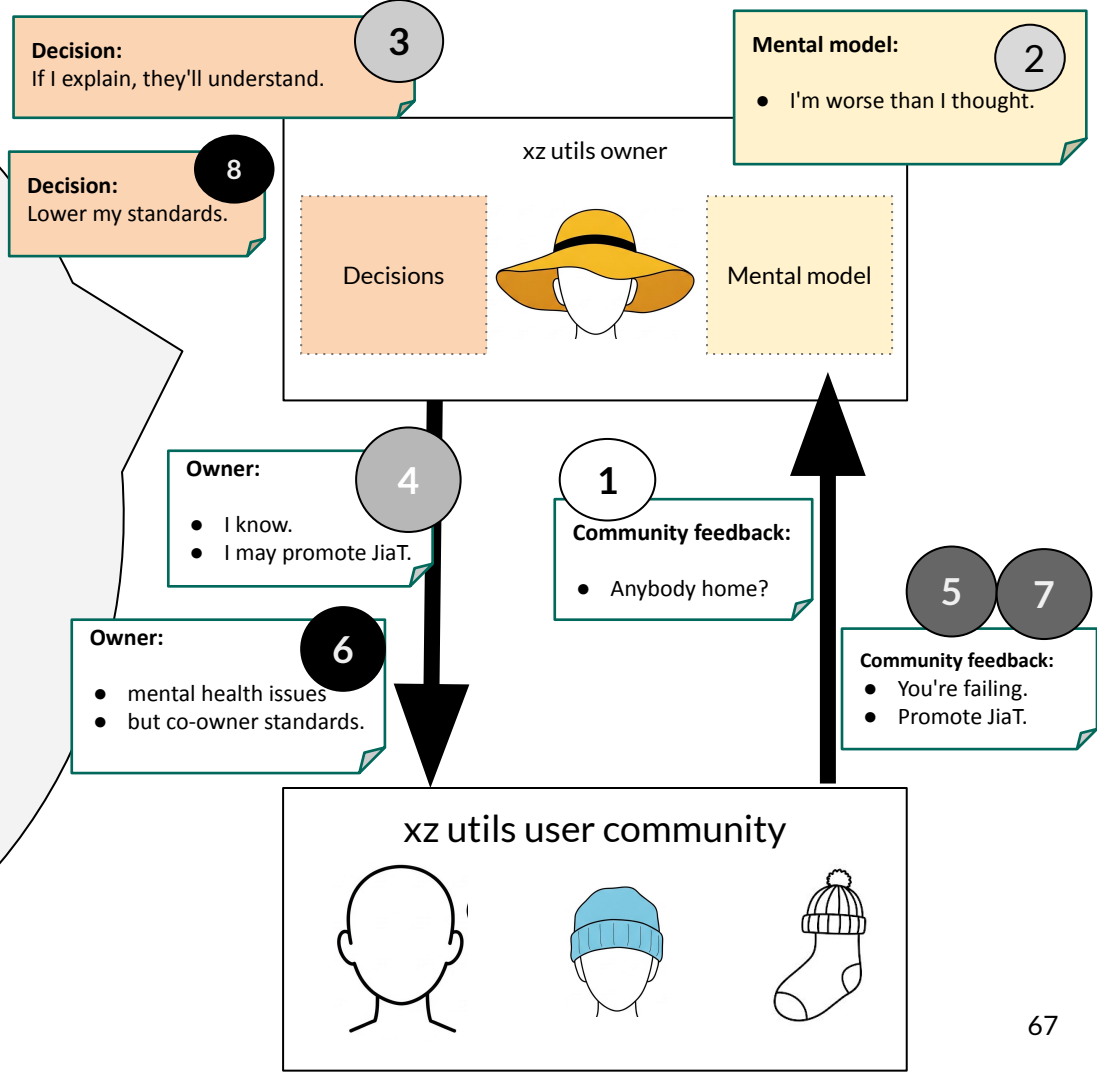
Please...?

Suggestion area

Blessed code

Everything happened over emails:

1. In May, a sock puppet asked whether part of the project was still maintained, noting unanswered email and a lot of inactivity.
2. Based on the owner's response the same day, we can say that the owner felt guilty about this, and also...
3. ...that he thought if he explained himself and expressed regret and good will, everyone would understand.
4. So he did. In a long email he explained that xz-utils was a side project he had little time for, that he could only fix bugs, and he ended with: It's clear that my resources are too limited (thus the many emails waiting for replies) so something has to change in the long term. ... Jia Tan might have a bigger role in the future.
5. A few weeks later, a new sock puppet went on the attack, telling the first sock puppet not to bother waiting, and saying: “Progress will not happen until there is a new maintainer. ... The current one has lost interest or doesn't care anymore.”
6. The owner replied immediately, admitting to mental health issues but also noting important reasons and standards for why he couldn't just promote Jia Tan.
7. In response, the other sock puppets just laid into him more, attacked him more.
8. And from what happened in a later control loop, we can conclude that the owner decided to lower his standards for making someone a co-owner.



[Original conversation](#) ([Google Doc copy](#))

Loop #2: Shoving xz utils into Debian

The next loop is how Jia Tan used sock puppets to force his backdoored version of xz-utils into Debian, past Debian's defenses.

?

xz-utils → Debian

We just need one slide first:
How does ***any*** package (like xz utils) get into Debian?

Debian release process

bit.ly/debian-infographic

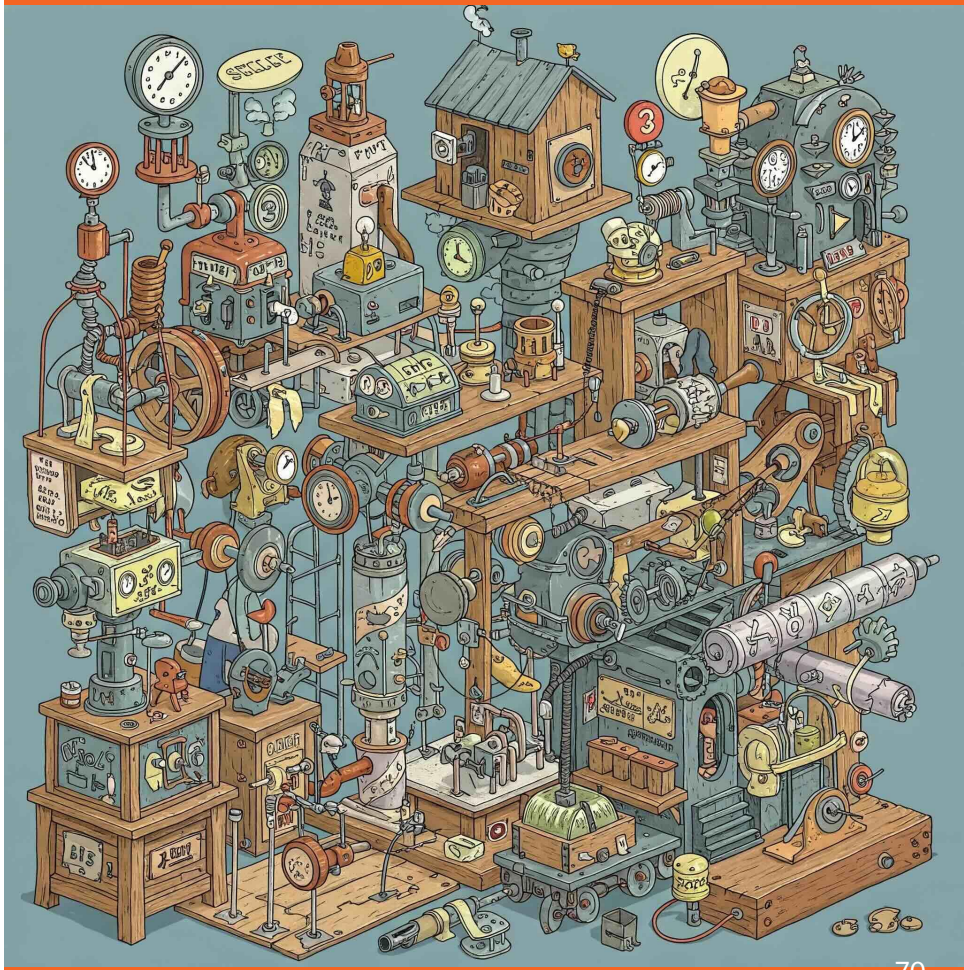
It's a process a few hundred volunteer developers have evolved over the decades, mostly to catch bugs before a release.

This is an AI-generated image, because due to image licensing, I can't show you the actual Debian infographic, only its link.

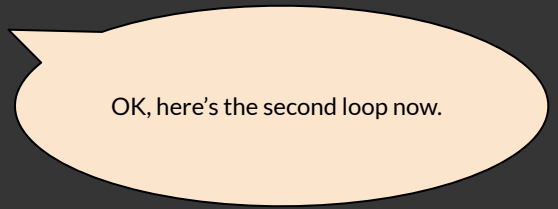
But the image captures how complicated the process is.

What that means, of course, is that it is hard to analyze for hazards.

The good news is, I'm about to describe a main part of it — using a control loop.



Loop #2: Shoving xz utils into Debian



Actors

Debian Developers are all-powerful Debian gods. They can make any code changes they want. There are around 400 of them, each with decades of experience. It's very hard to become a Debian Developer: you have to prove yourself to them.

A **Debian Package Maintainer** is a volunteer, typically a Debian Developer too, who is responsible for a few 3rd party packages. Their duties include knowing the package authors, tracking the code, and uploading new versions into Debian without a review from another Debian Developer.



Debian Developer

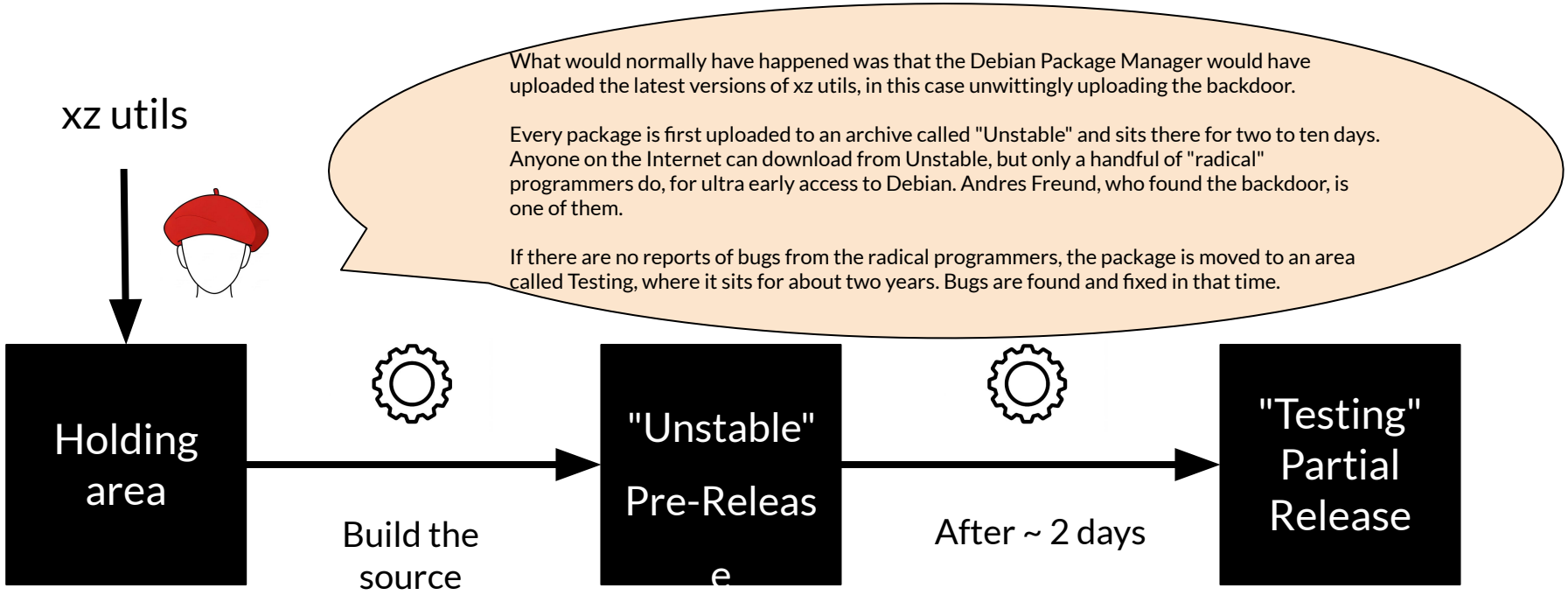


Debian Package Maintainer

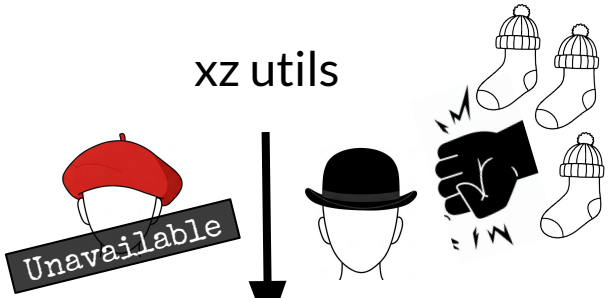


Sock puppet

What *should* happen



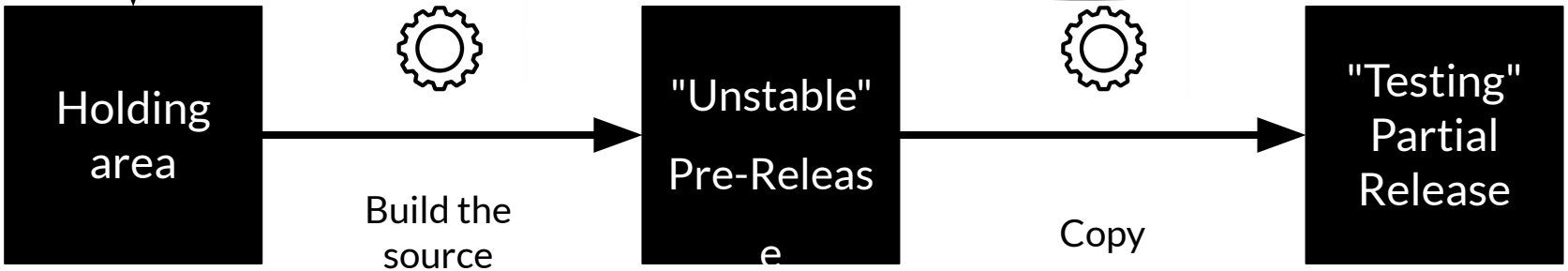
What happened:



What actually happened was that, for many reasons, the Package Maintainer wasn't available. So Jia Tan created a sock puppet on the Debian mailing list to request a so-called NMU — a Non-Maintainer Upload — of the latest version of xz utils.

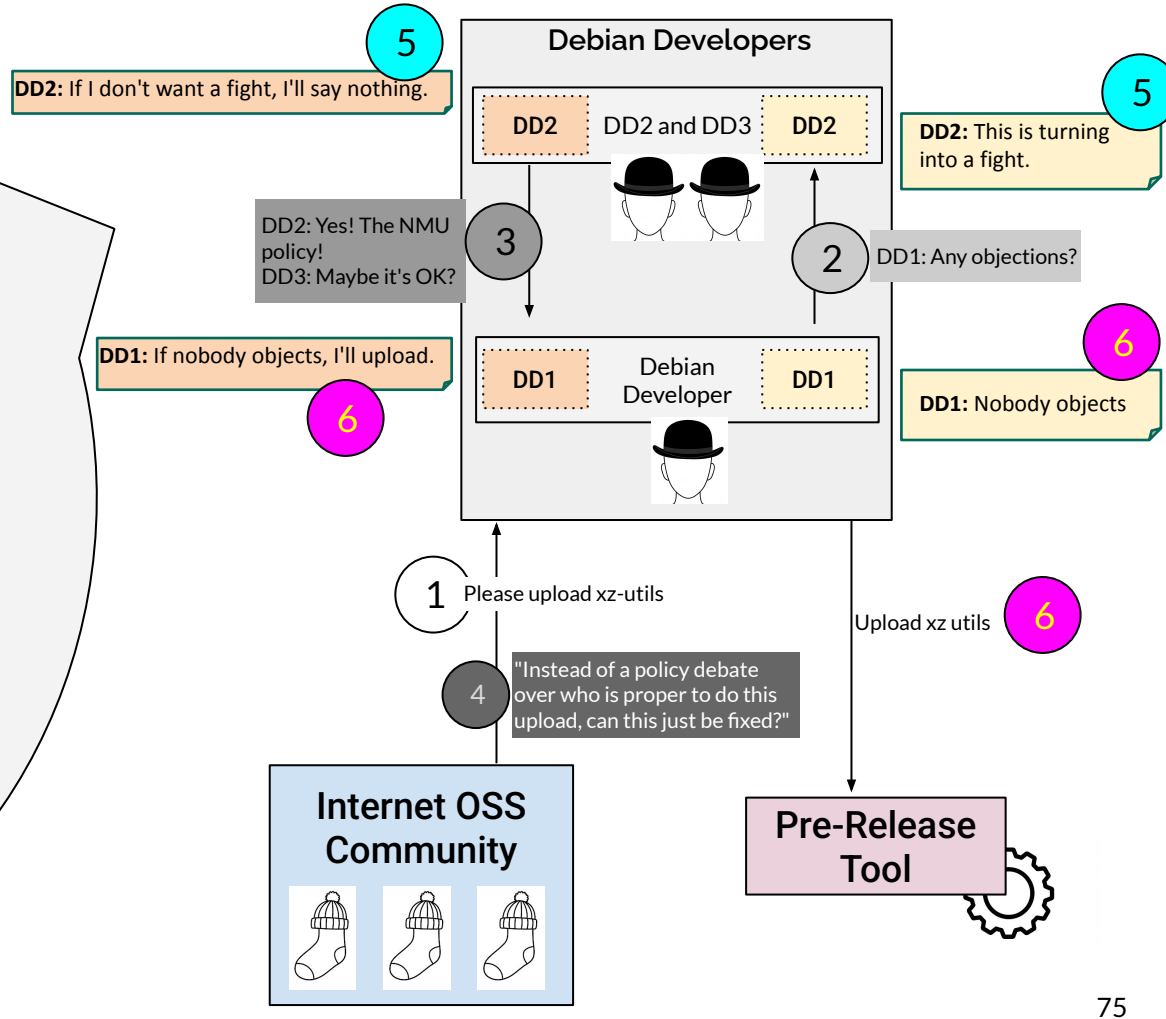
An NMU is when a Debian Developer who isn't the maintainer uploads a package. That's frowned upon, because typically that Debian Developer doesn't know much about the package. That's the Package Maintainer's job.

When a couple of Debian Developers raised objections to this NMU in principle, sock puppets bullied them into silence, and then a Debian Developer went ahead with the NMU to Unstable.



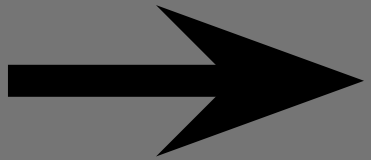
This is a control loop of how all that happened, again showing the algorithm and mental model of the involved controllers. **DD** stands for Debian Developer, the gods who can upload anything. (Note that we actually talked extensively to the **DD2** in this diagram, so we're not speculating about his thinking & mental model.)

1. On a Debian Developer mailing list, a sock puppet asked that any Debian Developer please upload the latest version of xz-utils,
   because it contained bug fixes.
   Other sock puppets emailed too, saying how happy they were for this fix to finally go in.
2. A Debian Developer DD1 started the upload process with a message everyone could read.
3. DD2 then chimed in, reminding everyone of the NMU policy, and asking: Where is the Maintainer?
   DD3 said he, DD3, had being stepping in silently for the Maintainer recently and knew about this package,
   so maybe an NMU would ok?
4. A sock puppet then started bullying by saying: "Instead of a policy debate over who is proper to do this upload, can this just be fixed?"
5. DD2 fell silent: he didn't want a big argument about NMU, missing maintainers, why policy matters, etc. All the other DDs fell silent too.
6. Hearing only silence, DD1 went ahead with the upload. The attack had succeeded.



Outline

1. Purpose of Analysis
2. CSD (Control Structure Diagram)
3. Scenarios
4. UCAs: actual & possible



5. Takeaways

Lessons

- OMG, so cheap!
- ~~Social engineering~~ Norm hacking

We learned two key lessons.

First, these control loop attacks cost virtually nothing—unlike pricey zero-days or two-factor breaches.

Second, they used what I'll call “norm hacking,” bending open source moral norms to manipulate behavior and bypass community standards.

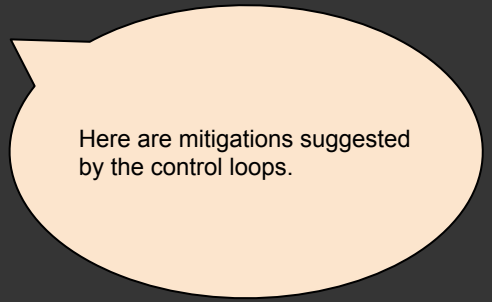
Researchable Hazards

- Single owner packages
- Linux release workflows
- STPA integrates security team insights

We found many hazards worth researching more.
Here are just three:

- How many packages have single owners? Actually, hundreds of thousands. How many are for critical infrastructure, like xz-utils was?
- How similar to Debian's are other Linux's release workflows? We haven't had time to look.
- In step 4 of STPA, security experts can use their strategies etc. to find possible causes of scenarios.

Mitigations




Up until the late 1950s, a businessman could bully a pilot into flying in bad weather — telling them they aren't a real pilot, that he'll tell other businesses not to not use them, and so on.


In 1958, the Federal Aviation Authority, the FAA, was created. After that, a pilot could just say: "Hey, them's the FAA regulations. I fly now, I lose my license. This conversation is over."
FAA
REGULATTION

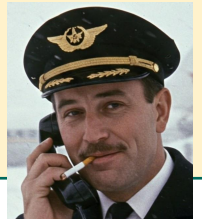
Decision:
Keep doing my thing.

xz utils owner

Decisions

Mental model

Mental model: I'm so cool







xz utils user community

Sock puppets

We want something similar for our control loops. For example, in the first one, we'd like the owner to say:

"Hey, Debian is happy with me only fixing bugs. And Debian won't take my code anymore if I quickly promote Jia Tan to co-owner. So please buzz off!"

Built into tools like GitHub, a warning could pop up if an owner tries to make someone a co-owner too quickly.

That's just one mitigation. Today Debian Developers propose and politely argue about many mitigations unmethodically—like, say, giving all packages two maintainers instead of one. With the control loops, we can find flaws in those mitigations they don't see. Our goal is that they do it without our help.

But the method is the main mitigation.

Example:

1. DDs propose mitigations
2. Control loop shows flaws
3. DDs fix mitigations
4. Rinse and repeat

Takeaways

1. We introduced a novel threat modeling method that reveals overlooked hazards, ...
2. ...requires no cybersecurity expertise, ...
3. ...and convinced a senior Debian Developer to use it for mitigation talks with his peers.

Yes: Our method is going to be tried in the field soon!

1. Novel method
2. ~~Security expertise~~
3. ~in active use

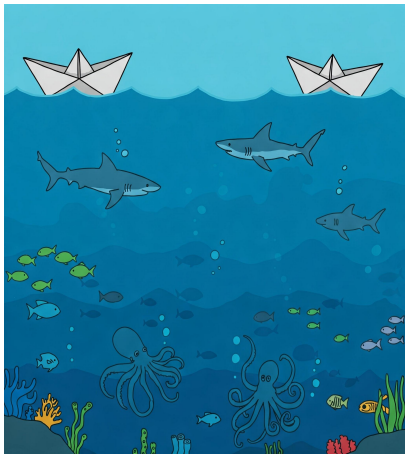

We've got a foot in the door — a first Debian Developers seeing the value of control loops, and bringing them to his colleagues. If that works, we'll try walking in — we'll show them STAMP, CAST, and STPA.
Stay tuned.

Image attributions

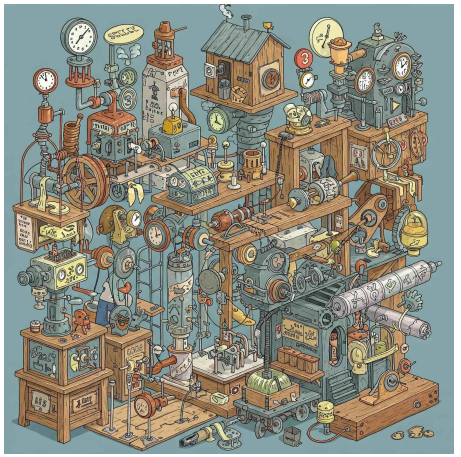
Generated with Gemini (gemini.google.com):

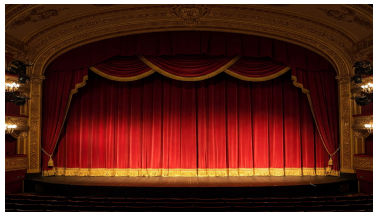


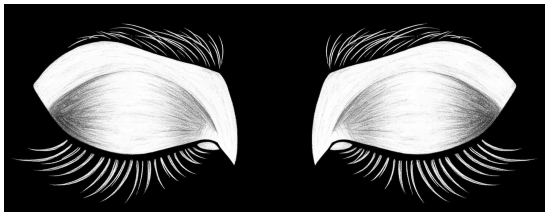










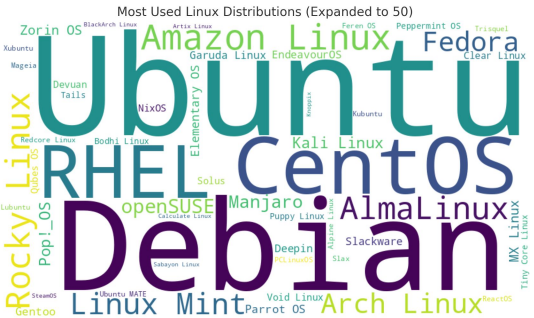






Generated with ChatGPT 4o (http://chat.openai.com)





Talk files:



Questions?

Problems? Email rbondi@google.com.