



# The Exploit Development Lifecycle

From **Concept** to **Compromise**

Valentina Palmiotti, Keynote,  
BSides Canberra 2024



# whoami?

- aka **chompie**
- Reverse **engineer**, vulnerability **researcher**, **exploit** developer, post exploitation **developer**
- Head of Vulnerability Research and Exploit Development, **IBM X-Force**
- **~7 years** in offensive security
- **Professional poster & Pwnie Award Winner**



weird machine mechanic



# Overview

The **What** and the **Why**?

1

## Misunderstood

Exploit development is typically misunderstood

2

## Bugs vs Exploits

0days get all the attention but it is really **exploit development** that **gives bugs their power**

3

## POC to Production

A lot of preparation is required for **production use-cases**

4

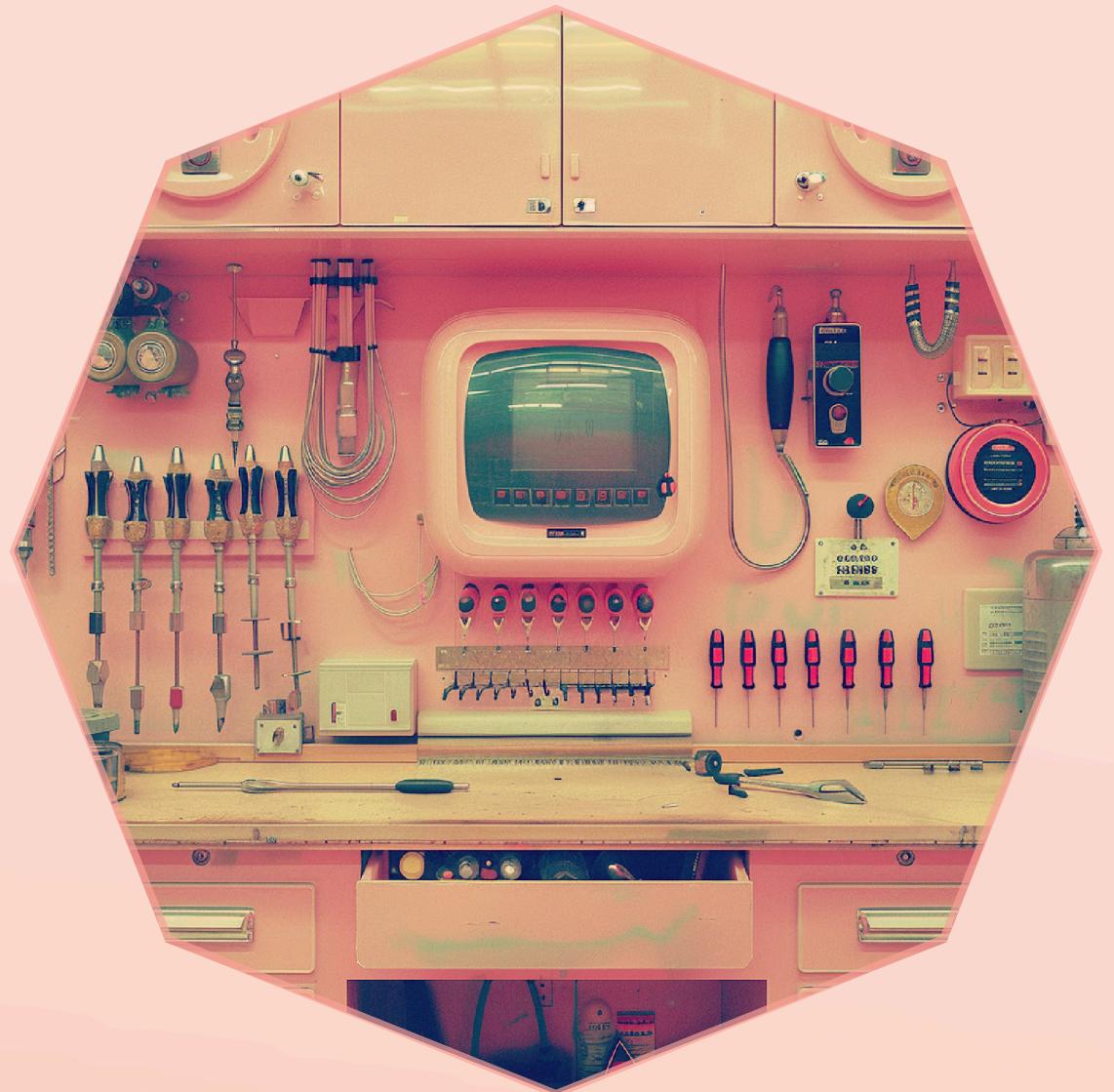
## Lifecycle

Unique **insights** into overseeing the **full exploit lifecycle**: From bug discovery to usage in sophisticated environments

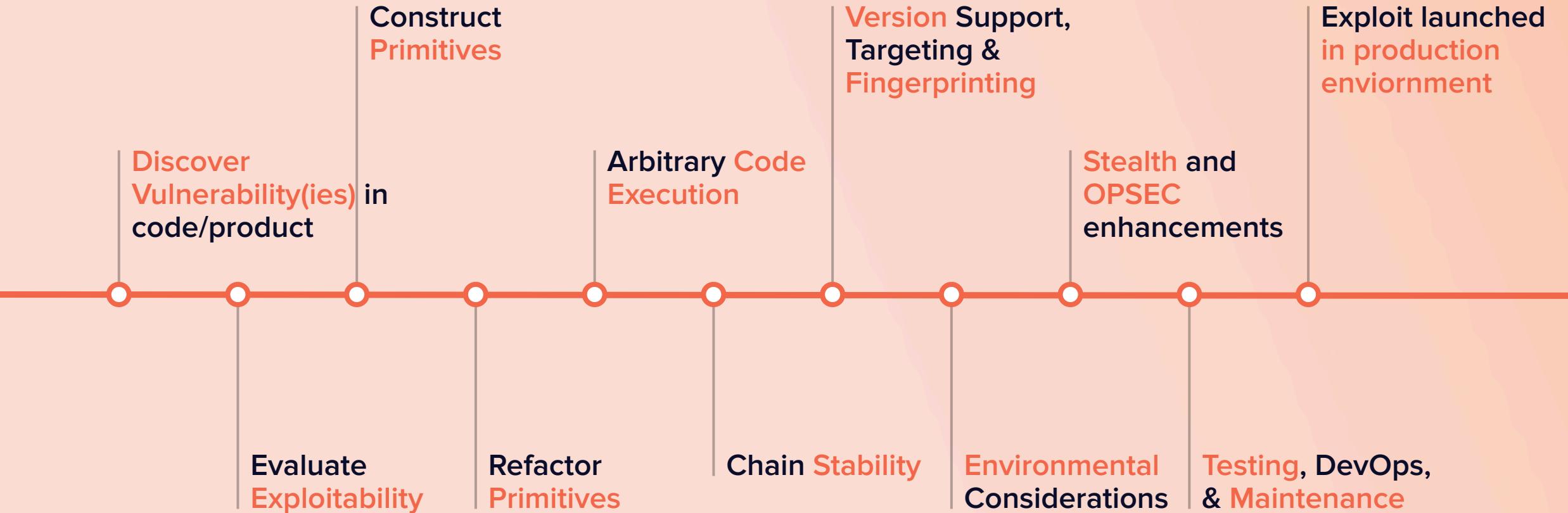
**“Exploit (n) : A method or piece of code that leverages vulnerabilities in software, networks, operating systems, or hardware to trigger behaviour not intended by the original designer or developer,,**

# Exploit are ~~Found~~ Built

- The **distinction** is important
- Finding a **vulnerability** is a **small part** of the **work**
  - Finding the bug
  - Analysing the platform
  - Availability of research labour
  - Doing the hard **development** and **testing** work
- This does not even factor in **specific target environment considerations** (e.g. EDR stack)
- Exploits are code, just like anything else



# Exploitation Timeline



# Not all Exploits are Equal

Various formats and degrees of **quality** for deliverables



I have a capability to can be **deployed** in a **variety of tested production configurations** with predictable outcomes

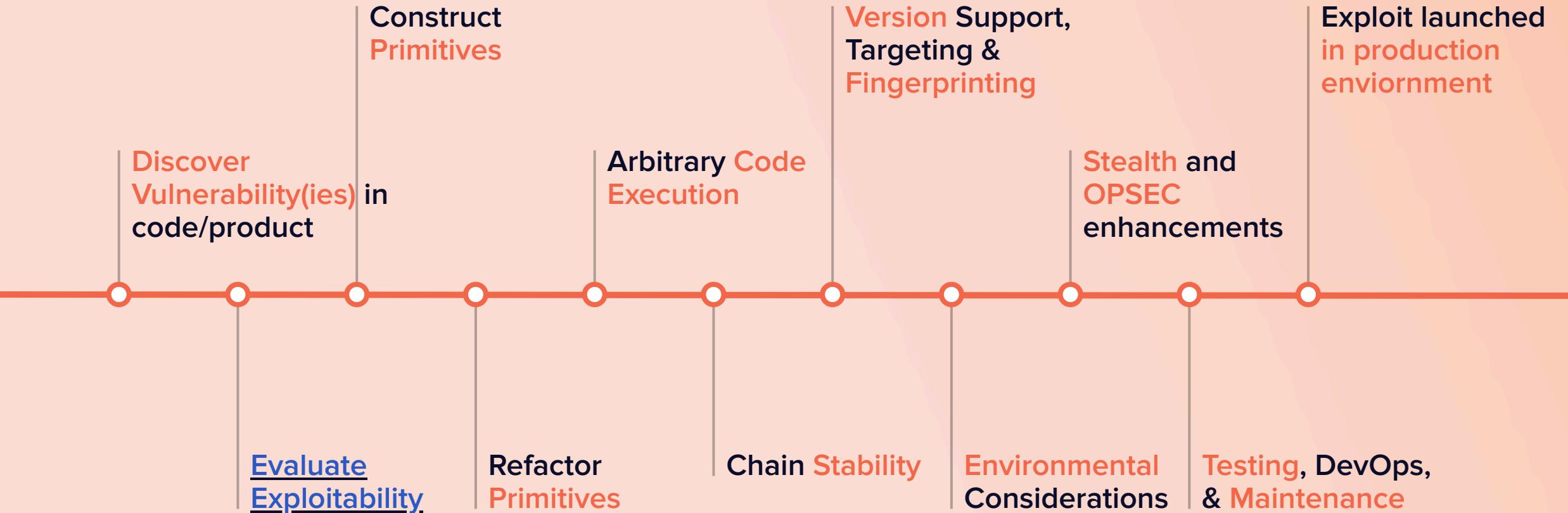


I can turn this behaviour into **arbitrary code execution (ACE)**



I have a bug and I can trigger **unexpected program behaviour!**

# Exploitation Timeline





# Evaluating Exploitability

# Good bug Bad bug

I promise this talk is not about bugs

- Software vulnerabilities produce exploit primitives
  - Primitives form generic building blocks for the exploit
- Vulnerabilities should be evaluated by the type of primitive they grant
  - Bad bug == Bad/Non-useful primitive

# Evaluating Exploitability: Initial Primitives

Is the bug a good bug?

- Vulnerabilities are **non-uniform**
- Many variables determine the usefulness of the primitive
- The variables involved **depend** on the **nature of the vulnerability**

# Vulnerability Properties

Some things to consider

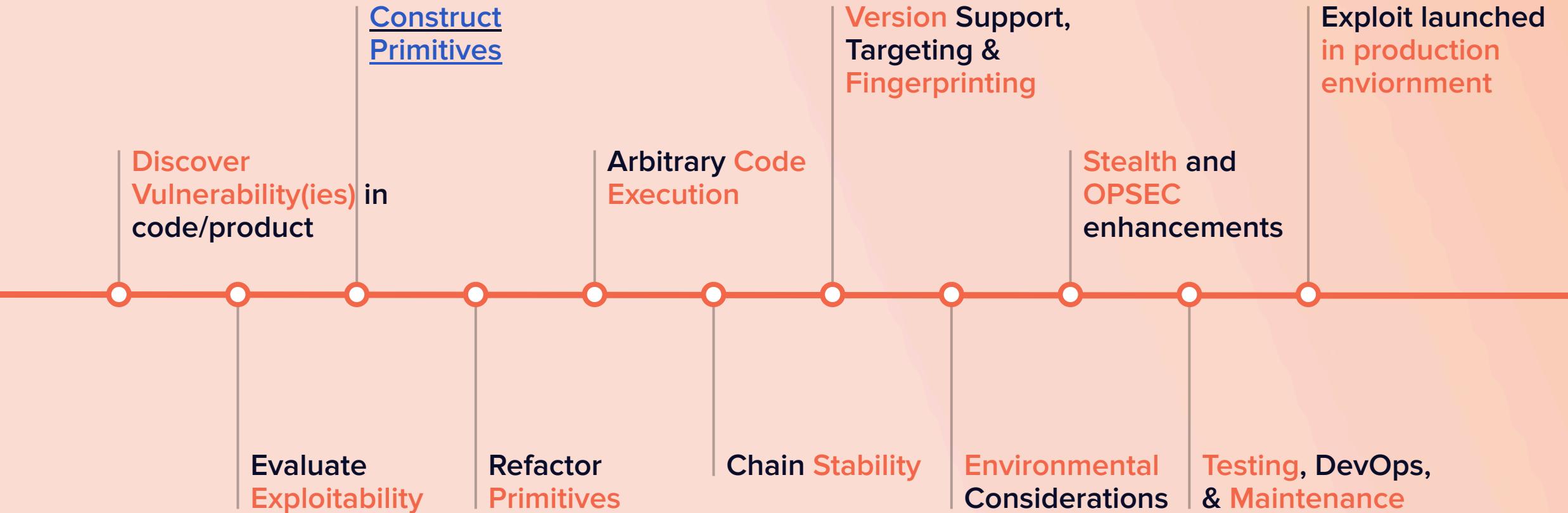
- **What can I influence, where, and how?**
  - Bit flipping at a static offset is less useful than a controlled write at a controlled offset
  - Some weaker primitives can be turned into stronger ones (e.g. out-of-bound read)
- **Repeatability**
  - Does this require **single-shot** exploitation **or** can the bug be **retriggered** to expand the primitive
- **Exploit Mitigations**
  - Don't **prevent bugs**, attempt to **thwart primitives**
  - May render seemingly **strong primitives** ineffective

# Evaluating Exploitability is a Cost/Risk Assessment

Betting research time against the reward

It is challenging to make definitive claims about non-exploitability

# Exploitation Timeline





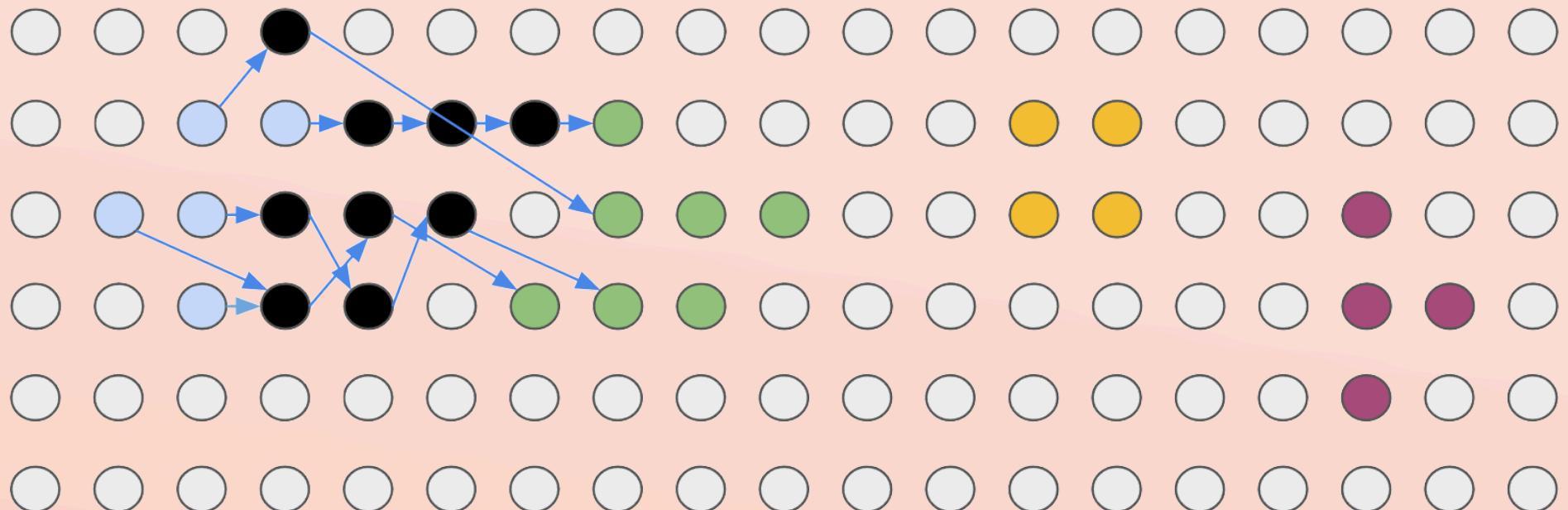
# Exploit Construction: Constructing Primitives

Triggering a **bug** causes the system to **transition from**  
**a normal machine** (execution follows only expected  
patterns) **to a weird machine** (execution follows  
patterns outside the specifications of the program)

# CPU State Space

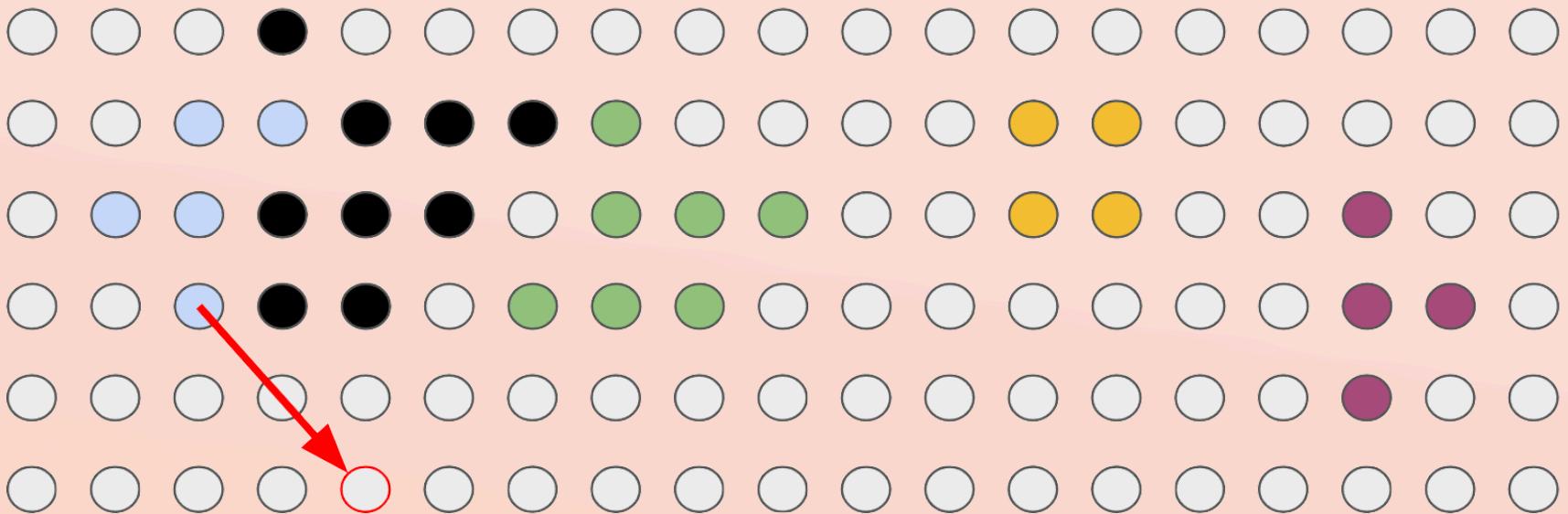
Limiting  
Weird  
Machines by  
Halvar Flake

CPU



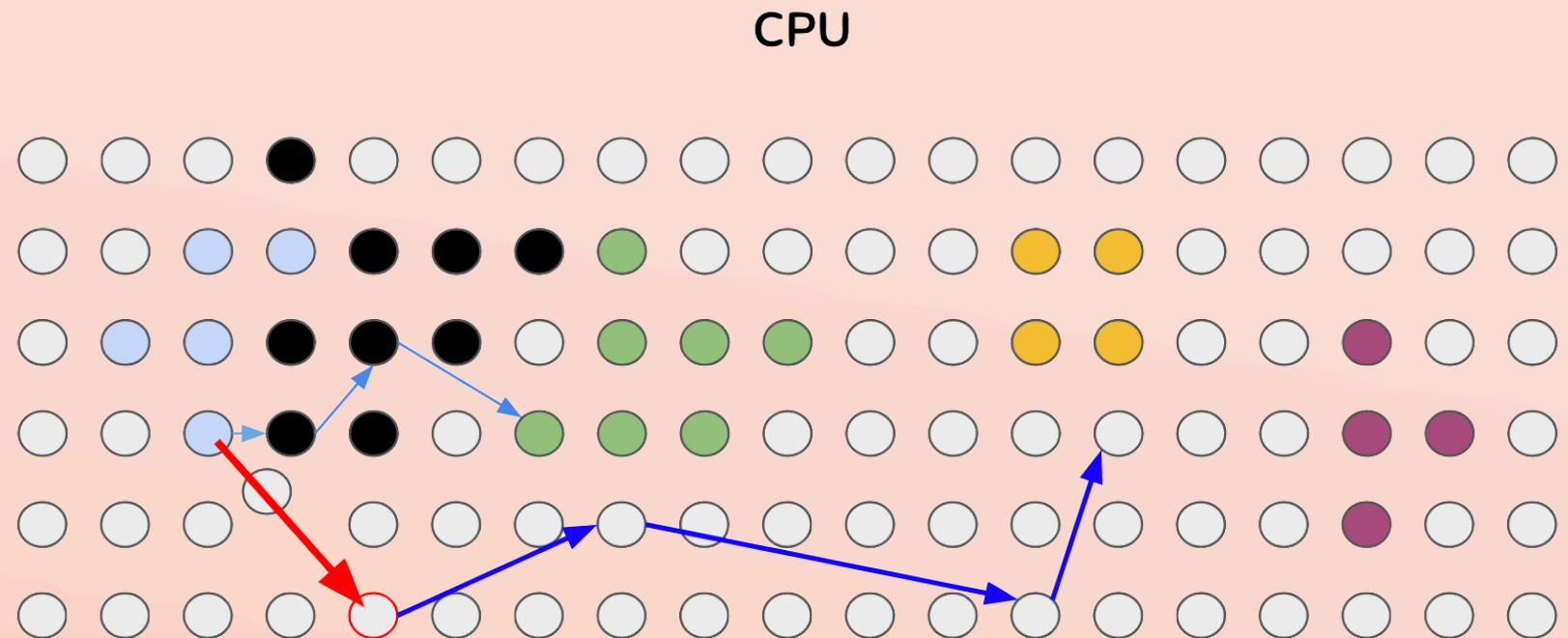
# Weird Machines

## CPU



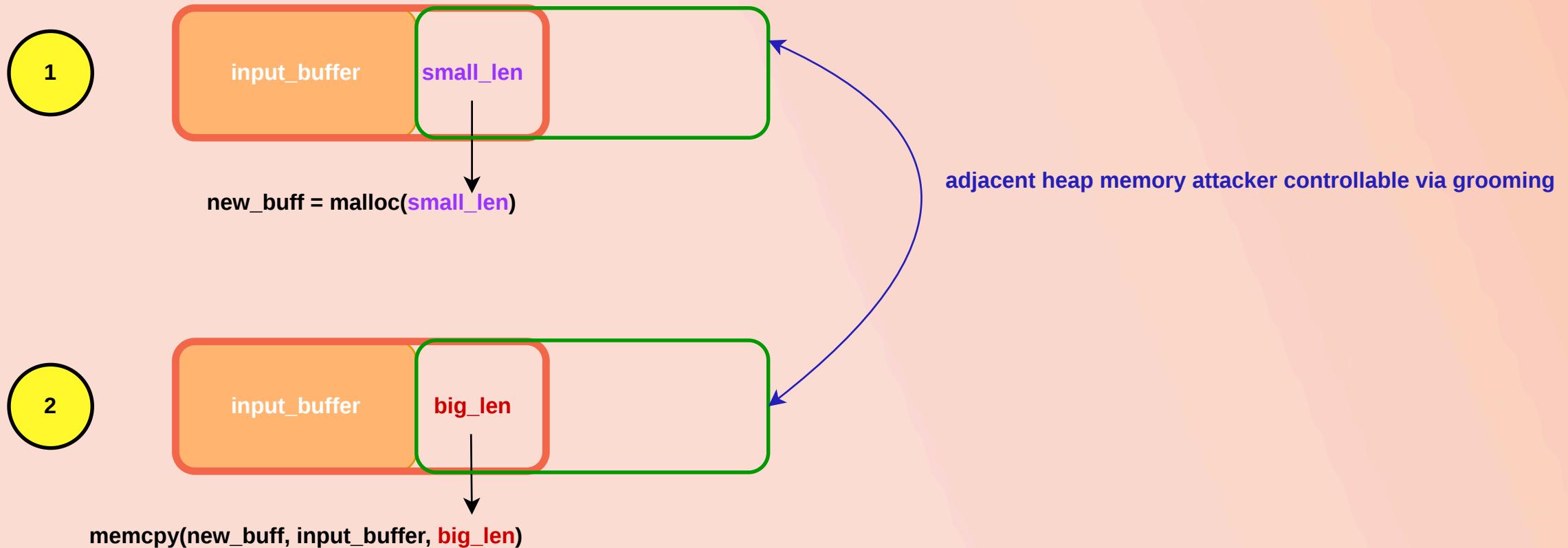
# Weird Machine Connectivity

- How close to Turing complete
  - What types of atomic operations can we perform using undefined application behaviour?

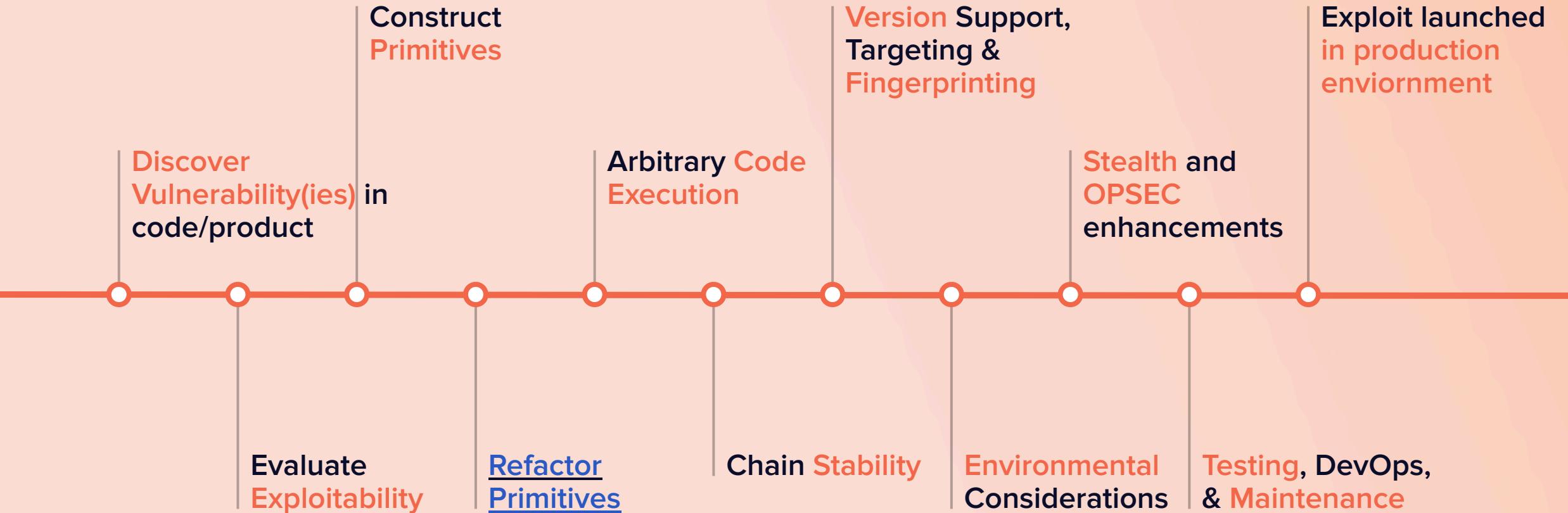


# Constructing Primitives

ex: Out of Bounds Read → Out of Bounds Write



# Exploitation Timeline

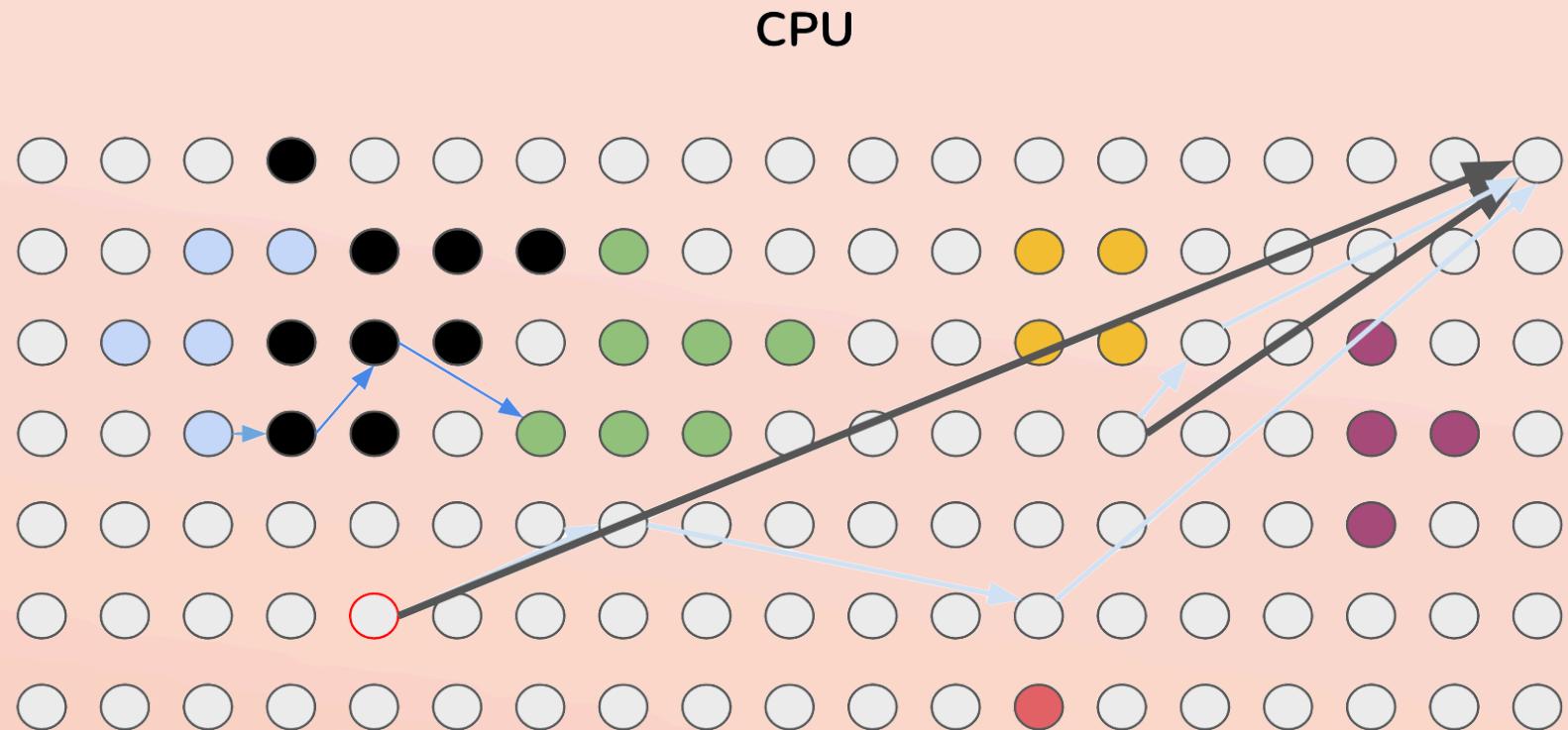




# Exploit Construction: Re-Factor(Construct) Primitives

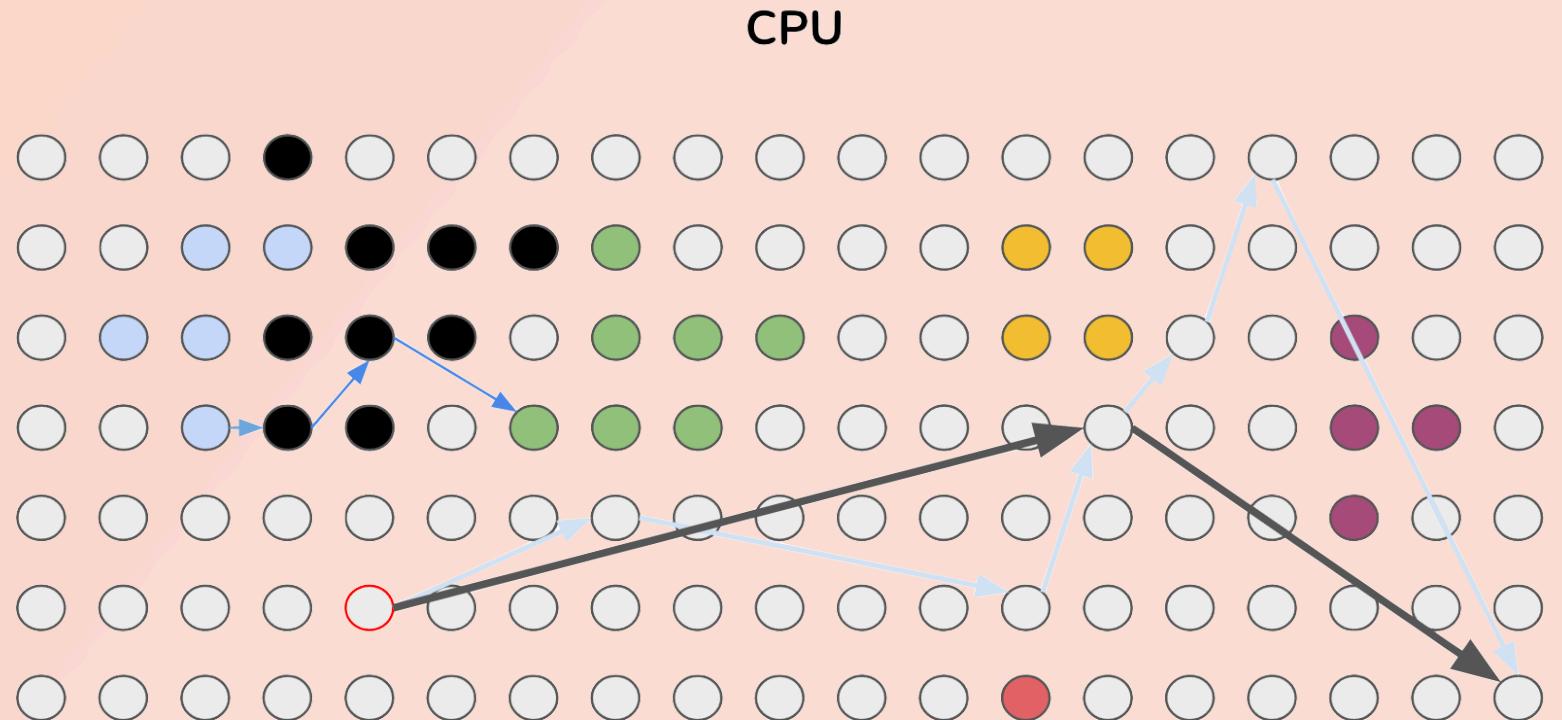
# Exploit Mitigations

- Reduce the number of reachable states from a weird state.
  - Ex: Address authentication mitigations bends many edges to SEGFAULT state

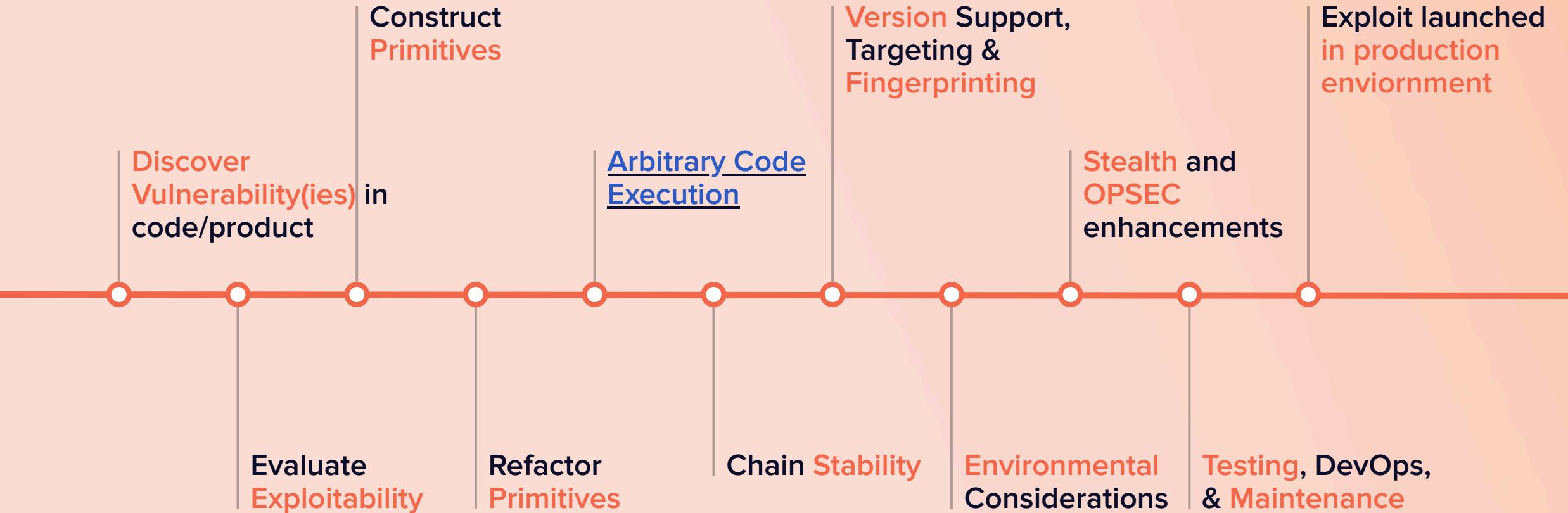


- Mitigations require **contortion of primitives**
  - Adds **additional steps** (state transitions) to achieve stable arbitrary code execution
- In some sad states, this could possibly be the end of the road
  - Why **Initial target assessment** is so important

# Mitigation Bypass



# Exploitation Timeline





# Exploit Construction: **Arbitrary Code Execution**

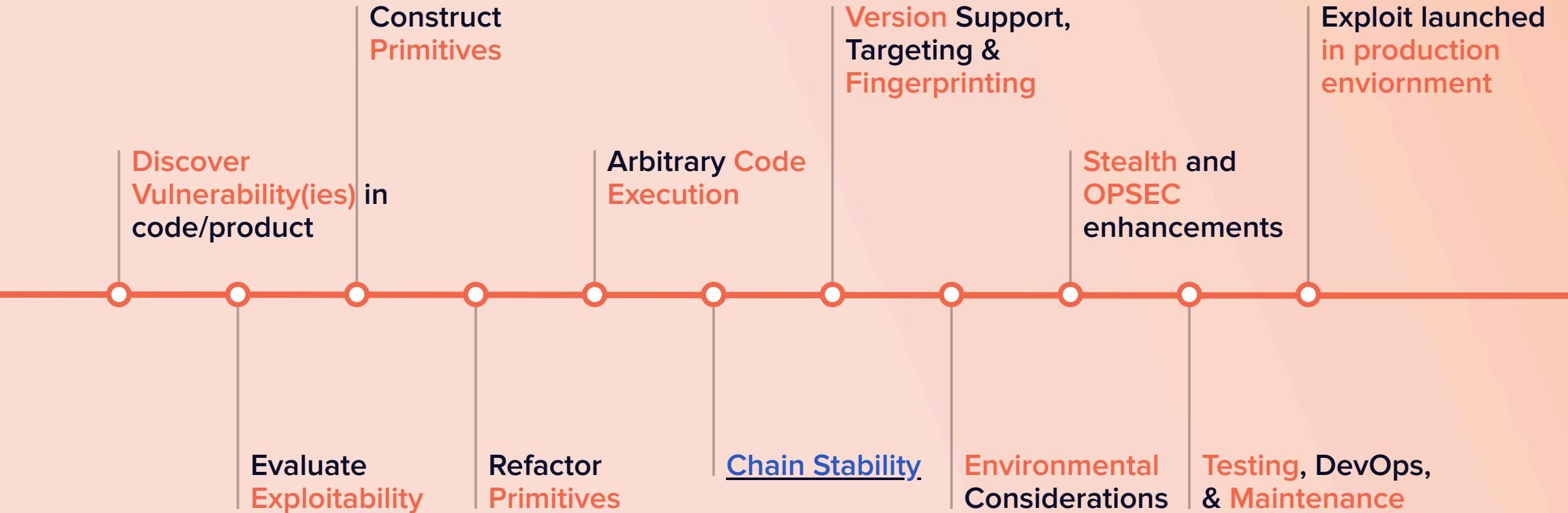
# Putting it all together

- Integrating exploit primitives together to achieve arbitrary code execution
- Multiple bugs can be chained together to supplement primitives
- Turing Complete Weird Machine

# Recap

- Bug or crash identified
- Viability assessment
- Build primitive chain
  - Rebuild chain as requirements/restrictions change
- Execute weird machine
- Arbitrary Code Execution (ACE)  Most public PoC's stop here

# Exploitation Timeline





# Exploit Construction: Chain Stability

**Exploits are not always deterministic**

Why are **live** exploit code **demos** so **tense**?



Things can **go wrong!**

# Exploit Reliability

- Sometimes depends on the nature of vulnerability
  - Could require getting system into a particular state to trigger vulnerable code path
  - Winning race conditions
- Mitigations can impact exploit stability
  - Some mitigations introduce randomness
  - Can introduce conditions that thwart individual exploitation attempts
- The goal is to increase exploit reliability to 100%
  - s.t. one trigger guarantees successful exploitation

# Methodology

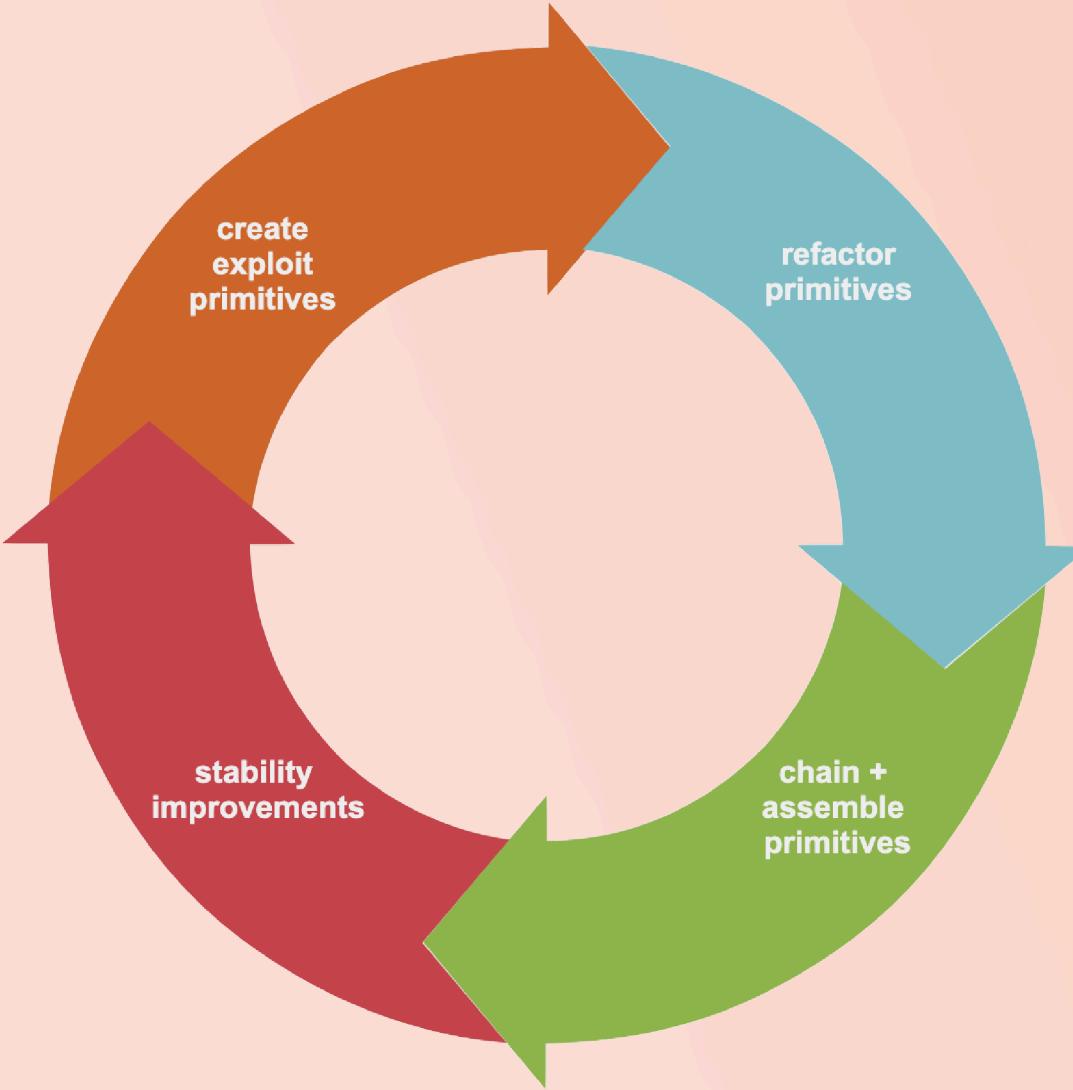
- Primitive Refinement: Examples
  - Tighten race windows
  - Add precision to sprays for vulnerable object allocation
  - Groom memory to predictable layout
  - Swapping objects in sprays
  - Find better program gadgets
- Developing new primitives
  - Layer primitives to create new capabilities
  - E.g. oracle primitives to bypass randomness mitigations
- Analyze program control flows + refactor exploit code to address the observed, known, possible weird machine states

# Clean-up

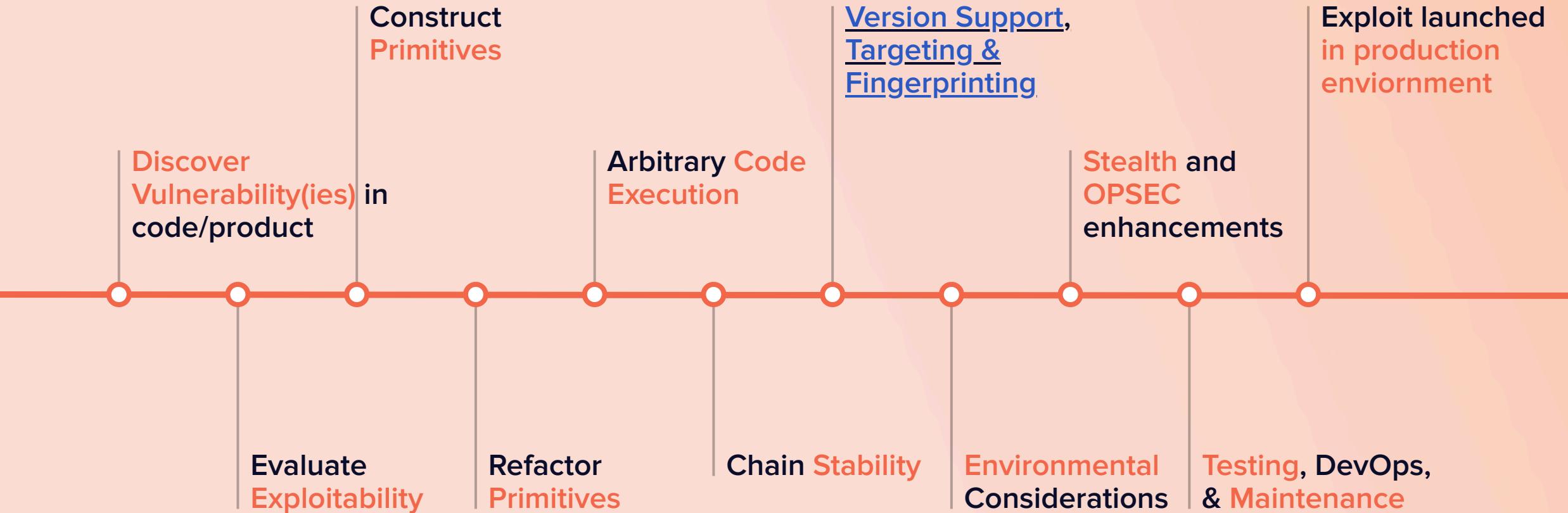
- **Release used resources**
  - Destroy objects
  - Free memory
  - Close handles
- **Is continuation a problem?**
  - Does the system behave normally, post exploitation?
- **Minimize exploit side-effects**
  - Restore to original pre-exploitation state (as much as is possible)

# Round and Round

The cycle  
goes on



# Exploitation Timeline





# Real-world Deployment: **Version Support**

# It works on my machine!

- The more targeted your code, the better
- Variances across targets can happen in many areas
  - Types of objects and object sizes
  - Implemented program features
  - Offsets, signatures
  - Available gadgets
  - Drastic differences in program logic
  - Implemented/supported exploit mitigations
- Broad production targeting means you have to deal with more variance



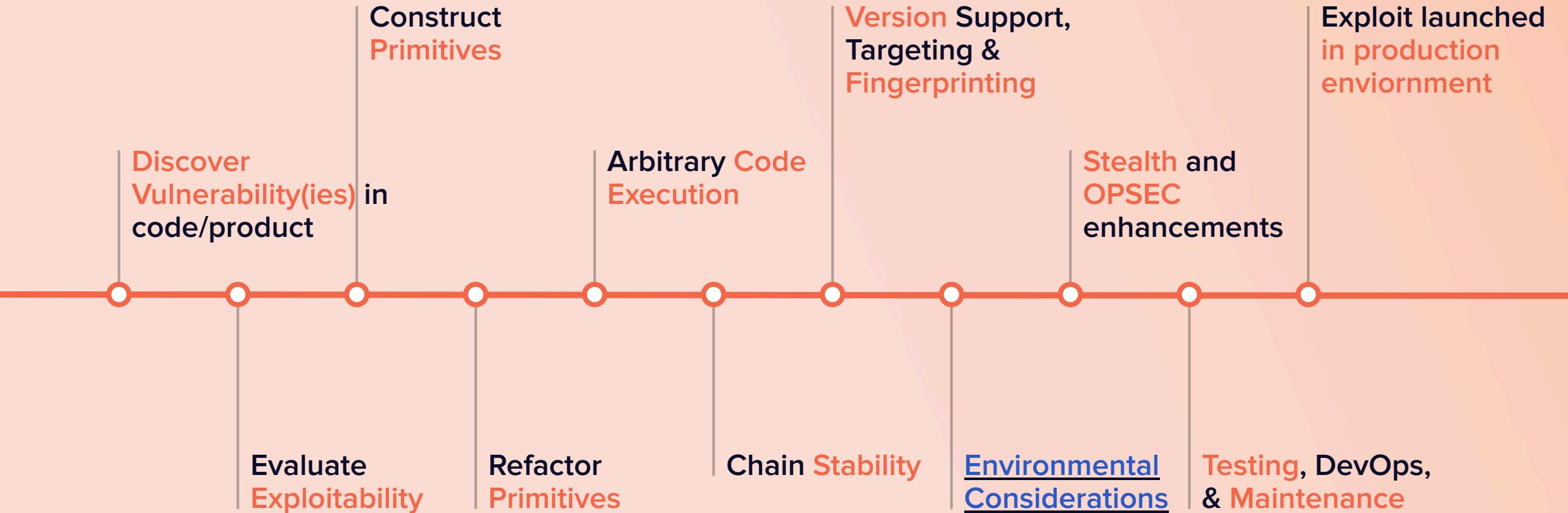
# Real-world Deployment: Targeting & Fingerprinting

# Situational Awareness

Is the exploit self aware?

- Targeting engine
  - Local fingerprinting
  - Remote fingerprinting
- Exploitation may vary not just based on software and OS version but also on hardware
- How will the exploit know which version of the chain should be used?
  - In-memory pattern matching
  - API call return information
  - System files (e.g. file build version numbers)
  - Service response metadata
  - Staged deployment?

# Exploitation Timeline





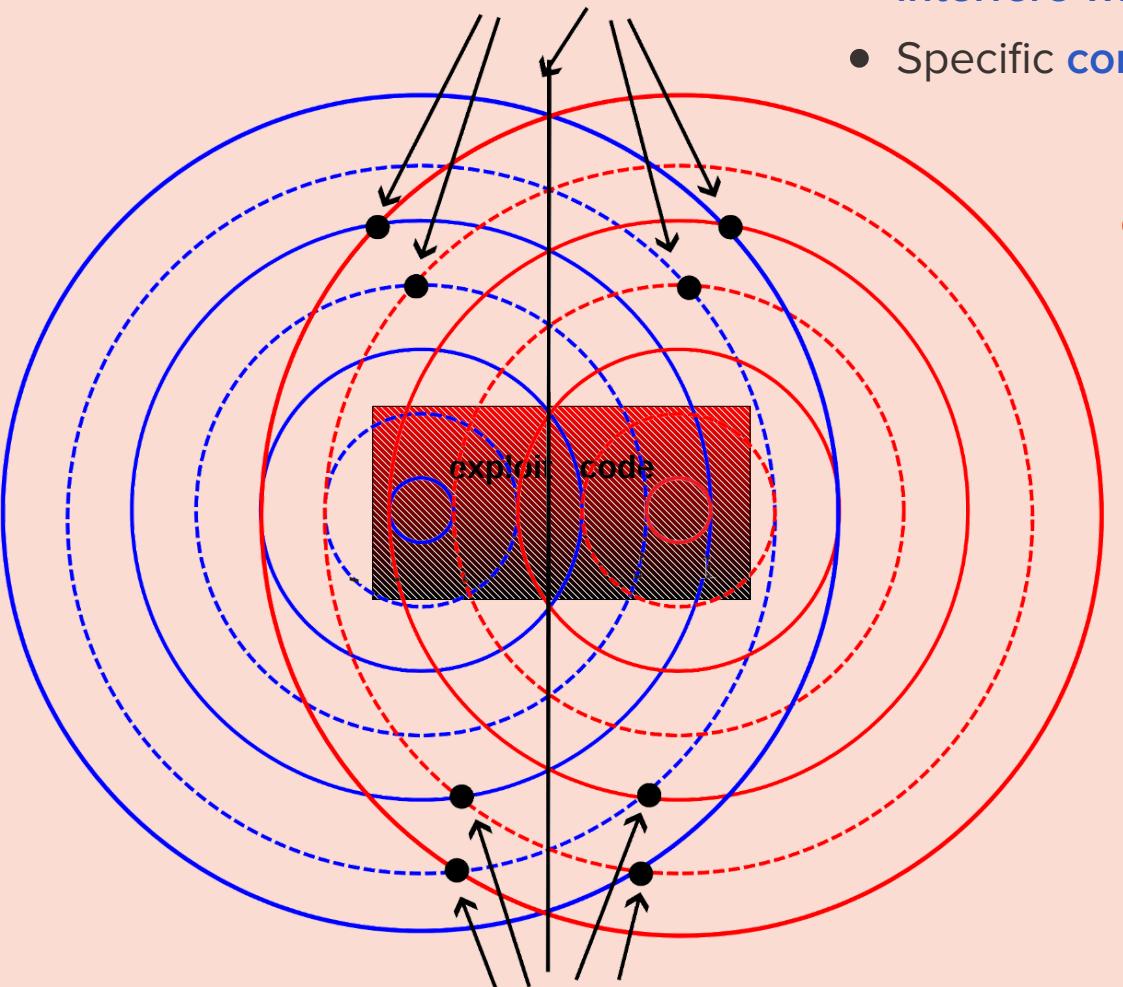
# Real-world Deployment: Environmental Considerations

# Exploits in the Real World

A lot can go wrong

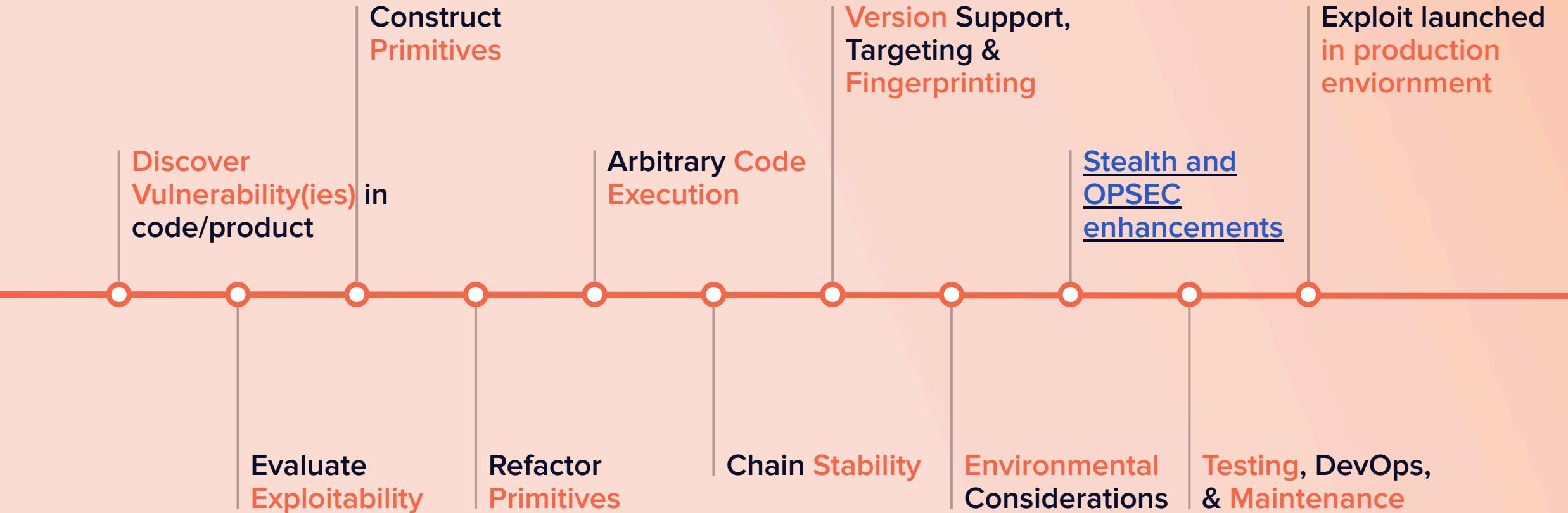
- Real life systems are hard to emulate
- The more details you have about the target the better
  - By closely emulating the target environment can anticipate what can go wrong beforehand
- Can try to approximate real systems by creating inorganic system noise
  - performance testing tools, stress tests in development kits

# Exploits in the Real World



- System load can affect reliability
  - Network traffic and running programs can use resources that interfere with exploit
  - Specific configurations can also interfere
- Hardware, network traffic, system noise
  - Low resource environment vs High resource environment
    - E.g. successfully exploiting race conditions requires a minimum core count

# Exploitation Timeline





# Real-world Deployment: **Stealth Mode**

# Protect Your Work

- Relevant for **private chains**
  - Exploits that can be burned
  - Unlike **public disclosure** and PoC's or competitions like Pwn2Own and TianfuCup
- In The Wild exploits ↘ Detection Technology ↗
  - Need to blindly **anticipate detection strategies**
- **Knowing your target is crucial, know what you're up against**
  - **EDR labs** and **simulated environments**
    - Offline Labs are tricky
    - Specific EDR configurations can impact detection/behavior
  - Valuable targets may be monitored by **solutions not available to the public**

# Don't Get Caught

Other steps of  
x-dev cycle  
come into play  
here!

- **Stability, can the exploit crash?**
  - Crash dumps are crucial in forensics, they could give the bug away completely
  - Clean-up, does the system run normally after exploitation? Does it release all resources?
- **Exploitation primitives, are they known?  
Signature?**
  - Bugs triggers by themselves and are hard to anticipate, consequently less likely to get caught
  - Exploitation side effects easier to signature, consequently more likely to get caught
    - Credential overwrites, token swapping, function table pointer overwrites, changing values in important memory
- **Environmental considerations, what is running on the target?**
  - EDR, other adversaries

# Don't Get Caught

Other misc.  
considerations

- **Size?**
- **Execution Method/Source?**
  - executable format?
  - folder path of exploit
- **Methods + Prep?**
  - calling suspicious APIs?
- **Strings?**

# Bottom Line

- Minimize the unavoidable side effects of exploitation
- Apply forensic counter-measures
- Make it like you were never there



# Real-world Deployment: **OPSEC** Attribution

# Not Like Stealth

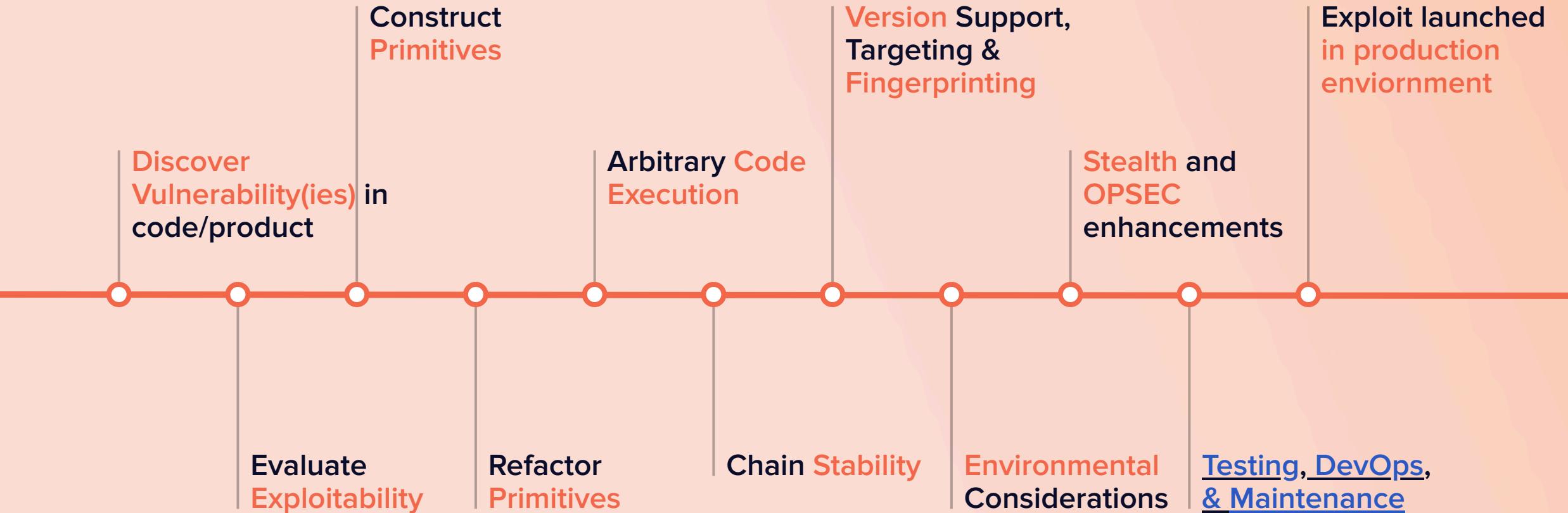
- Stealth is about hiding as best as possible, OPSEC is not so straight forward
- More in line with damage control
- Design decisions at this stage largely depend on the actor and their goals
  - Stakes can be high
  - Certain approaches can be better for some situations vs others
    - Subjective, almost an art form

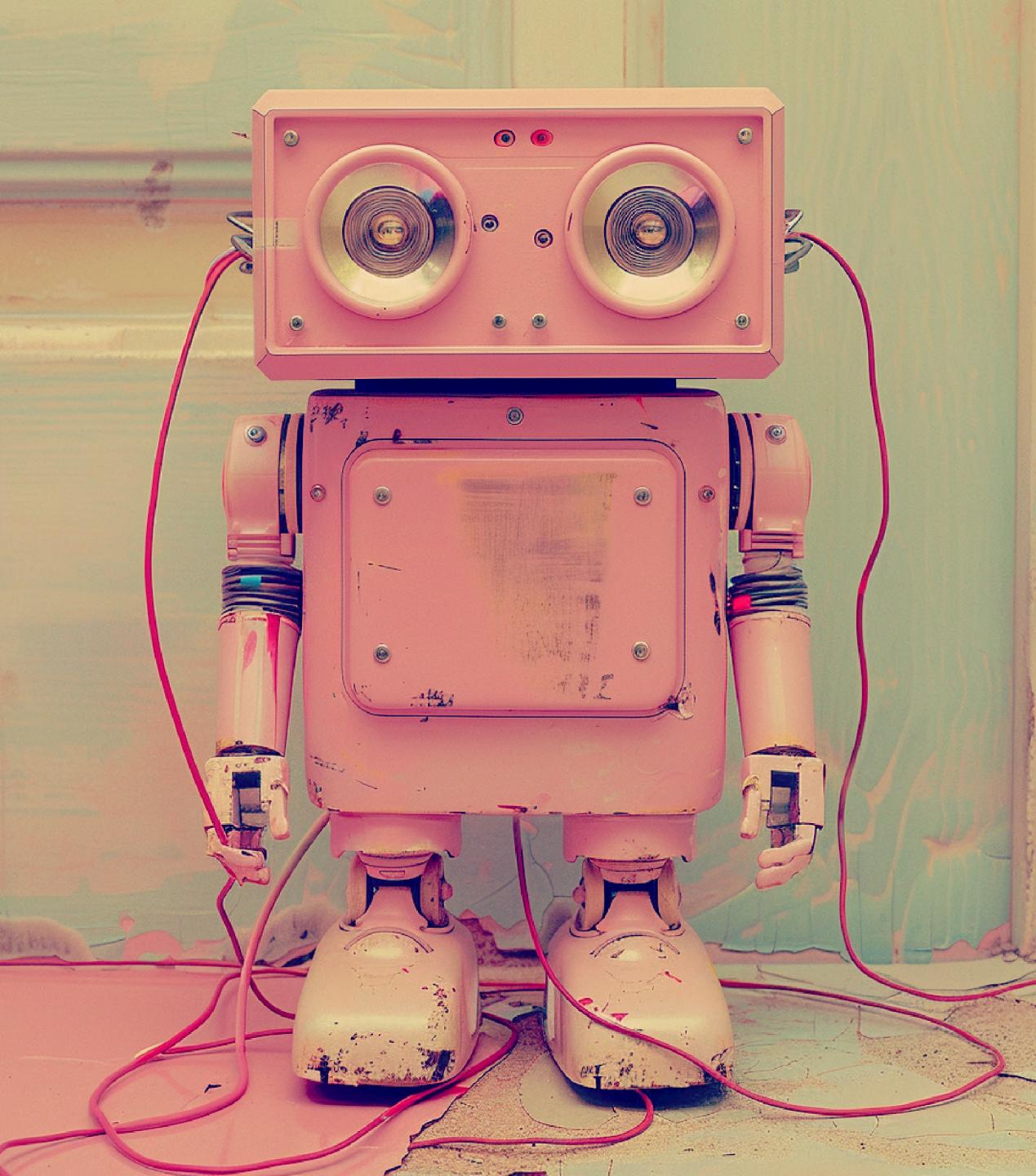
# Art

Strategy without tactics is the slowest route to victory.  
Tactics without strategy is the noise before defeat.

- **Binary artifacts** can inadvertently leave behind **metadata** (e.g. from compilers)
  - To **strip or not, or fake?**
  - Manipulation to **shift attribution?**
- **Anti reverse engineering & obfuscation**
  - If captured, make it **hard to analyse, to reproduce, or understand**
  - OR: make it look **innocuous**
- **Techniques & coding styles**
  - **Specific techniques**, primitives and code styles may be associated with a known **threat actor**
- **Minimize what can be burned**
  - Maximizing **novelty** isn't always the **best strategy**
  - **Separate post-exploitation** tooling from the **exploit chain**

# Exploitation Timeline





# Testing, Maintenance, & Exploit DevOps Considerations



# Exploits Are Software

- Proper Documentation so operators know how to use it
- Test against all targeted versions & common environmental constraints
  - Unit testing exploit primitives
  - EDR lab
  - Automation and sophisticated DevOps - attackers with more resources have an advantage
- Fixing the toolchain when unexpected situations inevitably pop up
- Keeping up with new OS/software builds
  - Offset custodians 😓

# Conclusion: Hug Your Neighborhood Exploit Developer

- Exploit development is **its own art**, separate from vulnerability research
- A lot goes into weaponizing a bug
  - More targeted, less ambiguity → less guesswork for development
- Complexity of SDLC make it so actors with **more resources** have a **big advantage**
  - Sophisticated exploit development is **highly skilled work** and very niche, **very few are doing this** for fun or for free
- The bar is high, which is why we rarely see **mass exploitation** of the **critical vulnerabilities** found in **valuable targets**
  - Some **exceptions** in exceptional circumstances (e.g. **EternalBlue**, leaked nation-state tooling)

# Thank You!

Halvar Flake

Mark Dowd

FuzzySec

wintercoats

# The End

Questions?

<https://chomp.ie/>

