Modeling the exploitation and mitigation of memory safety vulnerabilities

Breakpoint 2012

Matt Miller
Microsoft Security Engineering Center

Acknowledgements	Richard Tuffin, Julien Vanegue	For their insights and collaboration on classifying memory safety and exploitability

Related work	Automated/semi-automated exploit generation [1,2,3,16]	Thanassis Avgerinos, David Brumley, Sean Heelan, Edward Schwartz
	Memory safety classification & abstract models for exploitation [9,10,11,12,13]	Patroklos Argyroudis, Sandeep Bhatkar, Eep Bhatkar, Sergey Bratus, Daniel C. Duvarney, Halvar Flake, Michael E. Locasto, Chariton Karamitas, Meredith L. Patterson, Len Sassaman, R. Sekar, Anna Shubina, Ryan Smith, Chris Valasek
	Automated post-mortem analysis of exploitability [4,15]	Adel Abouchaev, Richard van Eeden, Nitin Kumar Goel, Damian Hasse, Scott Lambert, Lars Opstad, Andy Renk, Jason Shirk, Dave Weinstein, Mark Wodrich, Greg Wroblewski

Let’s start with three assertions

1. Assessing the exploitability of memory safety vulnerabilities is a hard problem

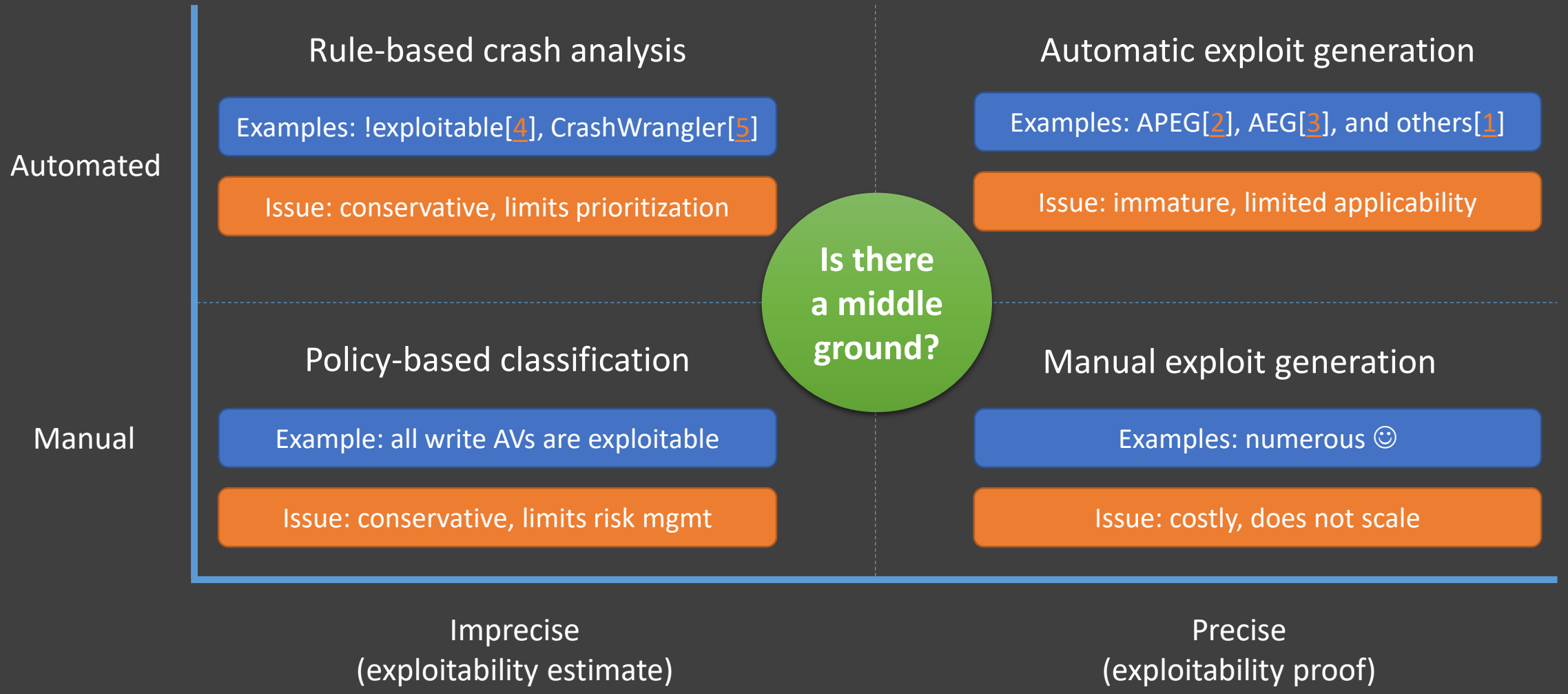
Today, an exploit must be written to prove exploitability (does not scale) or a conservative guess must be made (may overestimate)

2. There is no robust taxonomy for classifying the invariants of a memory safety flaw

CWE[7] can be used to classify a flaw’s type, but not its invariants (which influence the flaw’s exploitability)

3. You cannot improve on #1 without first addressing #2

Current methods for determining exploitability



Challenges in exploitability assessment

Software vendors must generally be conservative and coarse-grained when estimating exploitability

(!exploitable and Microsoft's Exploitability Index [4,6])

- Risk is often over-estimated[14]
  - Vendor must assume exploitable by default
  - Difficulty of exploitation must be generalized
- Reliant on manual estimation which is non-ideal
  - May be error prone, inconsistent, and hard to verify
  - Individuals must have exploit expertise or be conservative
- Hard to measure impact of mitigations (ex: DEP/ASLR)
- Limits patch deployment prioritization for vendor and customers

Researchers must agree with the vendor's assessment or concretely demonstrate exploitability

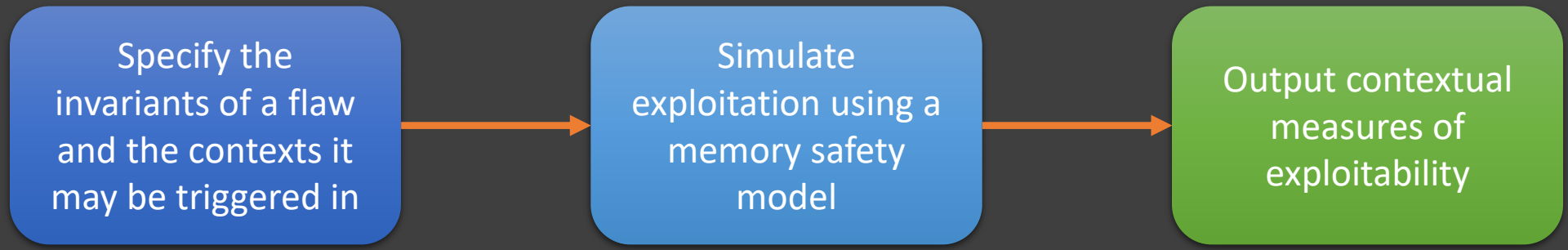
- May create tension between both parties
- May discourage reporting of legitimate issues

Proposing a way forward

To overcome these challenges, we need a method of assessing exploitability that is…

Accurate	The assessment is correct
Precise	Provides a granular assessment of exploitability that is aware of contextual factors
Objective	The assessment is consistent, repeatable, and can be independently reviewed
Scalable	Human involvement is minimized

An abstract model is one method that could be used to achieve this



Modeling memory safety

What is memory safety?

We have many terms for memory safety issues

- Buffer overrun, uninitialized use, type confusion, … [7]
- Read AV, write AV, write4, writeN, execute AV, …
- Temporal/spatial memory access errors[8]

+

We have many terms for memory safety defenses

- /GS, /SAFESEH, …
- DEP, ASLR, SEHOP, …

=

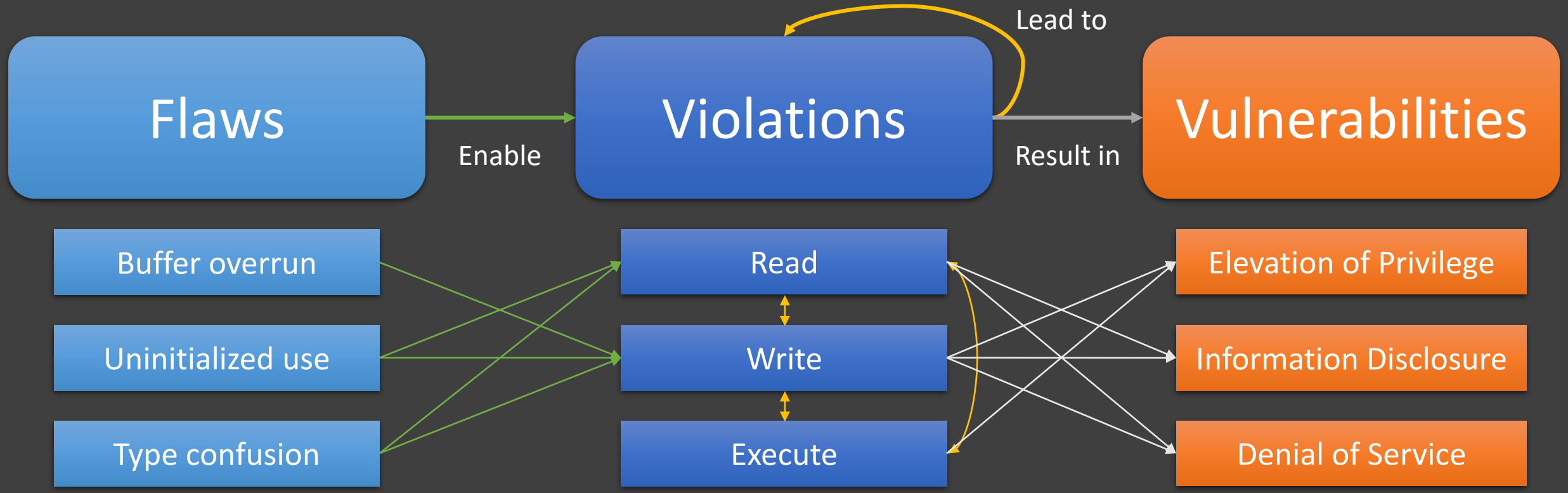
But how do they map to exploitability?

?

We need a more rigorous taxonomy if we want to be able to measure exploitability

Fundamental concepts in memory safety

Memory-safe: a memory access that is within bounds and refers to an object that is in a valid state



A flaw is the root cause of a vulnerability and directly enables one or more violation

A violation is an unsafe memory access that may lead to one or more subsequent violation

The maximum security impact of a flaw based on the violation(s) that are enabled

Classifying memory safety flaws

The taxonomy is generally agreed upon for memory safety flaws (e.g. CWE[7,8])

Arithmetic errors and other 2nd order issues can expose memory safety flaws

Boundary error	Type confusion	Uninitialized use
• Buffer overrun • Buffer overread • Out-of-bounds array index	• Invalid type cast • Invalid union field access	• Use after free • Double free • Uninitialized memory access

Temporal

Spatial

Each flaw can be mapped to the set of violation(s) that it enables

Classifying memory safety violations

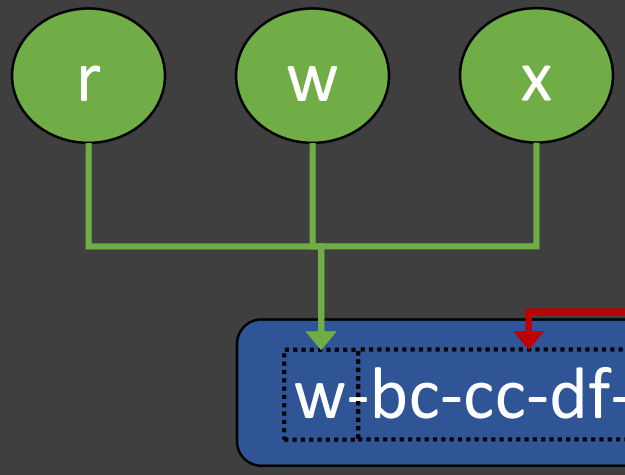
The properties of an unsafe memory access are a convenient way to describe a violation

Two properties establish the basic types of violations

Extended properties add more precision

Memory access method

Memory access parameter state



Parameter
Base (b)
Content (c)
Displacement (d)
Extent (e)

State
Controlled (c)
Fixed (f)
Uninitialized (u)
Unknown (?)

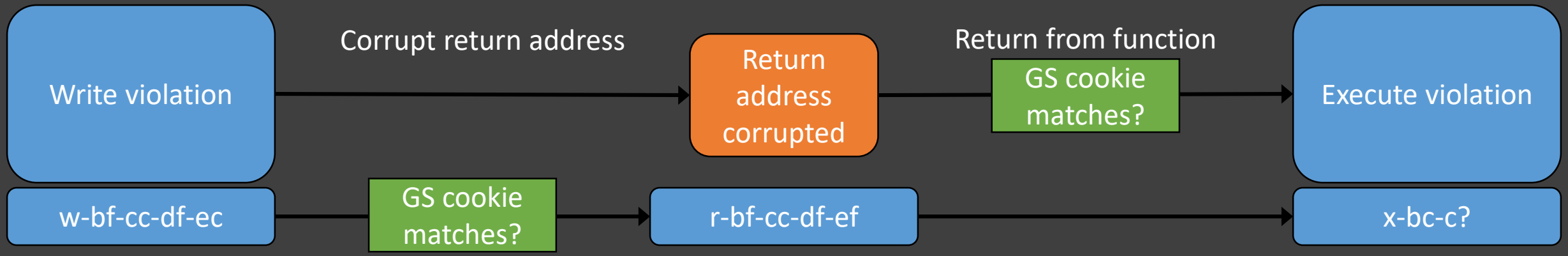
Property	Values
Addressing mode	absolute, relative
Direction	forward, reverse
Initial displacement	post-adjacent, pre-adjacent, …
Base region type	heap, stack, …
Execution domain	user, kernel
Locality	local, remote
…	

= Write, base and content are controlled, displacement and extent are fixed

Modeling exploitation & mitigation techniques

An exploitation technique enables a transition from one type of violation to another

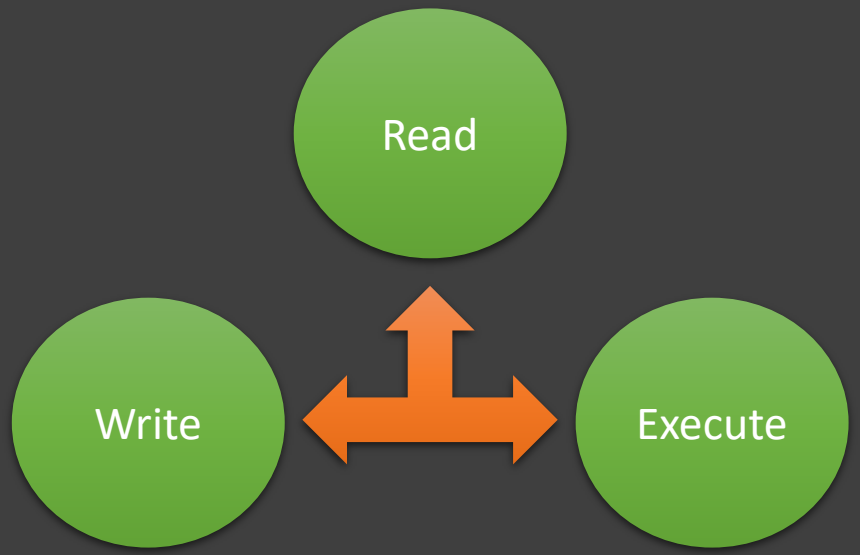
Canonical exploitation technique: stack return address overwrite



A mitigation technique introduces new constraints on these transitions

Exploitation primitives

There is a finite set of primitives for transitioning between the basic types of memory safety violations



Transition	Primitives
r → r	Read value used as base, displacement, and/or extent of a subsequent read
r → w	Read value used as base, content, displacement, and/or extent of a subsequent write
r → x	Read value used as base of an execute
w → r	Corrupt memory used as base, content, displacement, and/or extent of a subsequent read
w → x	Corrupt writable code and execute it
x → x	Execute with a controlled base and/or content

Exploitation techniques combine these primitives in different ways to reach a desired end state

Vulnerability classification

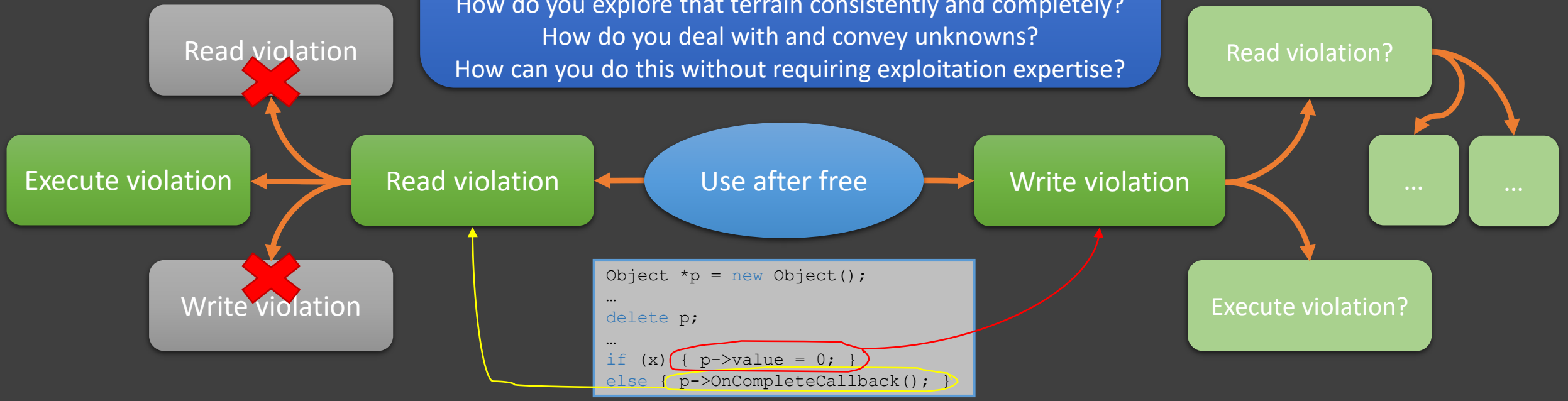
A typical vulnerability triage workflow

An uninitialized function ptr is read from the freed object leading to an execute violation. The read does not lead to a subsequent read or write.

Challenges with this workflow

How do you identify the "terrain" of violations to explore?
How do you explore that terrain consistently and completely?
How do you deal with and convey unknowns?
How can you do this without requiring exploitation expertise?

A constant is written to freed mem, but the subsequent violations are unknown as they depend on what is corrupted by the write.



1. Determine the root cause (the flaw)
2. Explore violations enabled by the flaw
3. Stop when exploitable violation found

VEXClass: a vulnerability classification assistant

VEXClass is a proof of concept tool that adds more structure to the vulnerability triage & classification process

Enables uniform and consistent classification

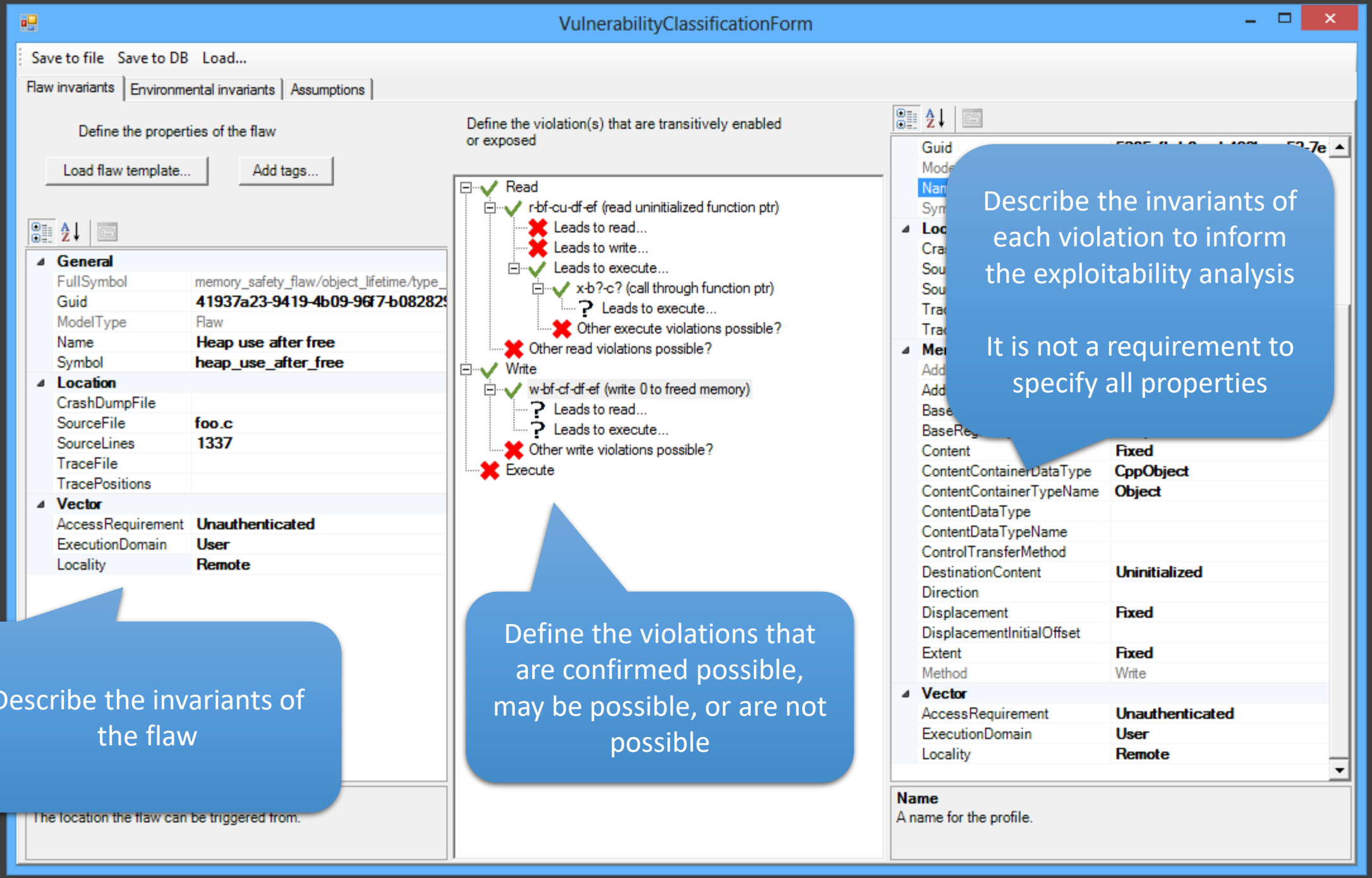
- Well-defined model for flaw and violation invariants
- Supports expression of “known” unknowns
- Classification output is normalized & easily reviewed

Provides a map of the terrain to explore

- Can be used as a “checklist” for the triage process
- Helps ensure that all possibilities are considered
- Analysts do not need state-of-the-art exploitation expertise

Facilitates more precise exploitability analysis

- Classification output is used as input for exploitability analysis
- Separates classification process from exploitability assessment
- Advances in exploitation do not require re-classification


VulnerabilityClassificationForm
Save to file
Save to DB
Load...
Flaw invariants
Environmental invariants
Assumptions
Define the properties of the flaw
Load flaw template...
Add tags...
General
FullSymbol
memory_safety_flaw/object_lifetime/type_
Guid
41937a23-9419-4b09-96f7-b08282
ModelType
Flaw
Name
Heap use after free
Symbol
heap_use_after_free
Location
CrashDumpFile
SourceFile
foo.c
SourceLines
1337
TraceFile
TracePositions
Vector
AccessRequirement
Unauthenticated
ExecutionDomain
User
Locality
Remote
The location the flaw can be triggered from.
Describe the invariants of the flaw
Define the violation(s) that are transitively enabled or exposed
Read
r-bf-cu-df-ef (read uninitialized function ptr)
Leads to read...
Leads to write...
Leads to execute...
x-b?-c? (call through function ptr)
Leads to execute...
Other execute violations possible?
Other read violations possible?
Write
w-bf-cf-df-ef (write 0 to freed memory)
Leads to read...
Leads to execute...
Other write violations possible?
Execute
Define the violations that are confirmed possible, may be possible, or are not possible
Guid
Content
Fixed
ContentContainerDataType
CppObject
ContentContainerTypeName
Object
ContentDataType
ContentDataTypeName
ControlTransferMethod
DestinationContent
Uninitialized
Direction
Displacement
Fixed
DisplacementInitialOffset
Extent
Fixed
Method
Write
Vector
AccessRequirement
Unauthenticated
ExecutionDomain
User
Locality
Remote
Describe the invariants of each violation to inform the exploitability analysis
It is not a requirement to specify all properties
Name
A name for the profile.

Example output

```
<VulnerabilityDescription xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
  <Flaw>
    <Name>Heap use after free</Name>
    <Symbol>heap_use_after_free</Symbol>
    <Guid>41937a23-9419-4b09-96f7-b0828290a3c4</Guid>
    <Locality>Remote</Locality>
    <AccessRequirement>Unauthenticated</AccessRequirement>
    <ExecutionDomain>User</ExecutionDomain>
    <SourceFile>foo.c</SourceFile>
    <SourceLines>1337</SourceLines>
  </Flaw>
  <Violations>
    <Violation>
      <Name>read uninitialized function ptr</Name>
      <Symbol>r-bf-cu-df-ef</Symbol>
      <Guid>a01d0f76-6b58-4d4d-9e9d-69f1782e9154</Guid>
      <TransitiveViolations>
        <Violation>
          <Name>call through function ptr</Name>
          <Symbol>x-b?-c?</Symbol>
          <Guid>744feefa-0fe4-4ebf-8eba-7610a36cda4f</Guid>
          <TransitiveViolations />
          <Method>Execute</Method>
          <BaseState>Unknown</BaseState>
          <BaseRegionType xsi:nil="true" />
          <ContentState>Unknown</ContentState>
          <ContentDataType xsi:nil="true" />
          <ContentContainerDataType xsi:nil="true" />
          <DestinationContentState>Nonexistent</DestinationContentState>
          <DisplacementState>Nonexistent</DisplacementState>
          <DisplacementInitialOffset xsi:nil="true" />
          <ExtentState>Nonexistent</ExtentState>
          <AddressingMode>Absolute</AddressingMode>
          <Direction xsi:nil="true" />
          <ControlTransferMethod xsi:nil="true" />
```

Normalized classification output enables information sharing and exploitability analysis

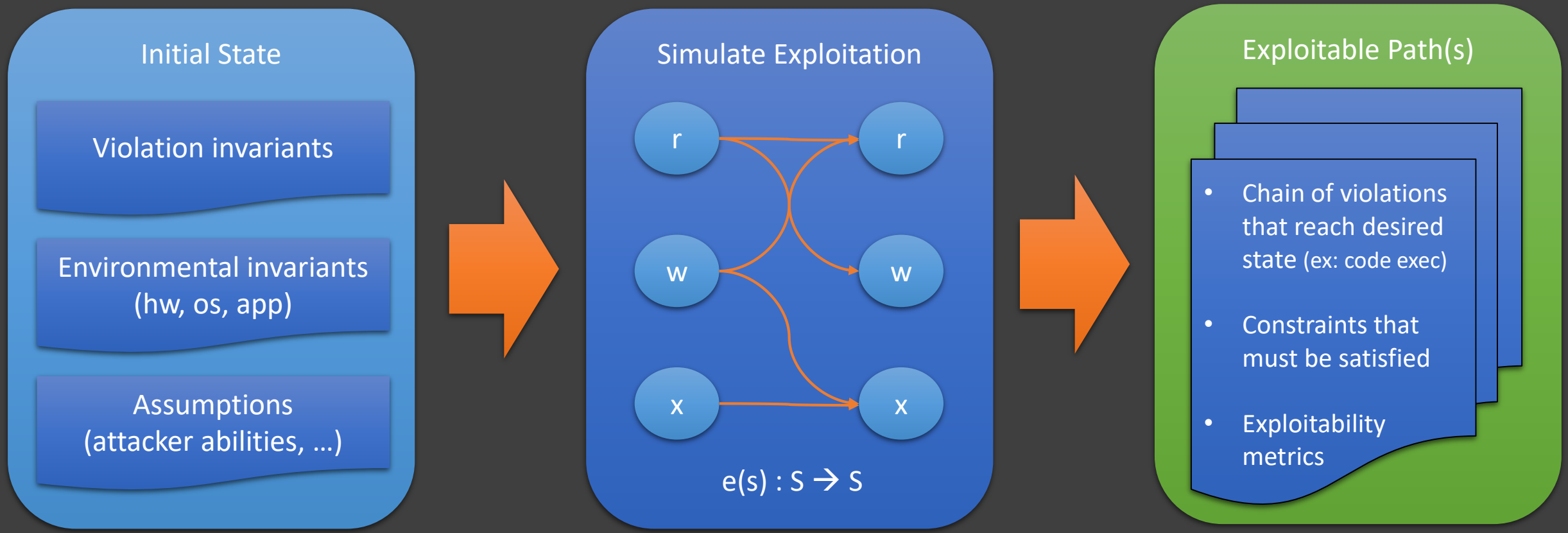
(without giving away specific details of the issue)

VEXClass Demo

Measuring exploitability

Measuring exploitability via an abstract model

The invariants of a violation form part of the initial state that is used to measure exploitability



This model quantifies exploitability based on the probability that a sequence of exploitation techniques will succeed

Simulating exploitation

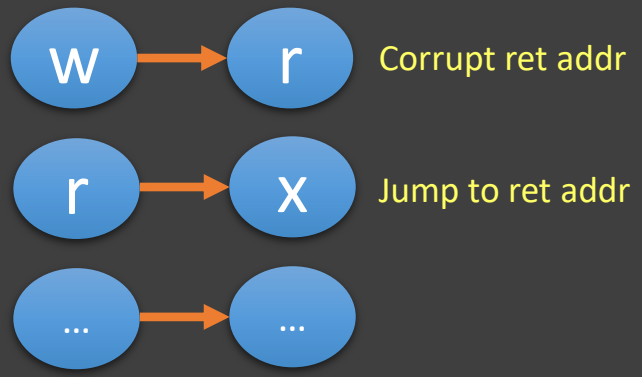
A state machine (NFA) is a convenient way to model exploitation

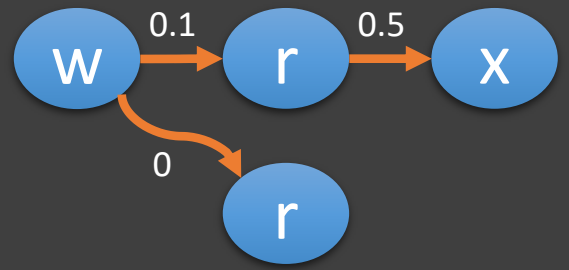
Each ***state*** is composed of the invariants of a violation and the context it is triggered in

Exploitation ***techniques*** define the transition function from one state to another

Each transition has zero or more constraints that can be ***probabilistically*** satisfied

Violation	w-bf-cc-df-ec
Application	Internet Explorer 9
OS	Windows 7 SP1
HW	x86 + pae
…	…



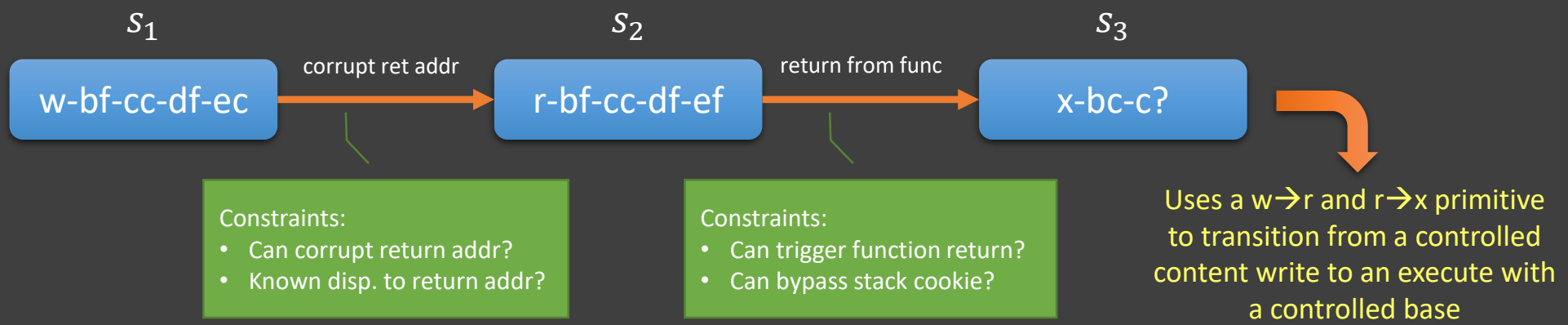


One technique is exploitable 5% of the time, the other not at all

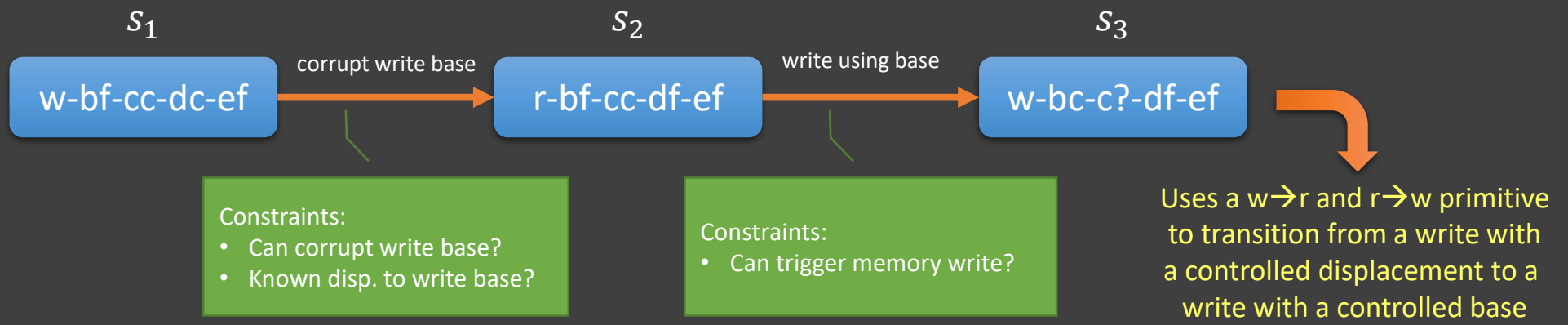
Simulation runs until reaching a fixed point $[e(s) = s]$ or a desired end state (e.g. code execution)

Examples of exploitation techniques

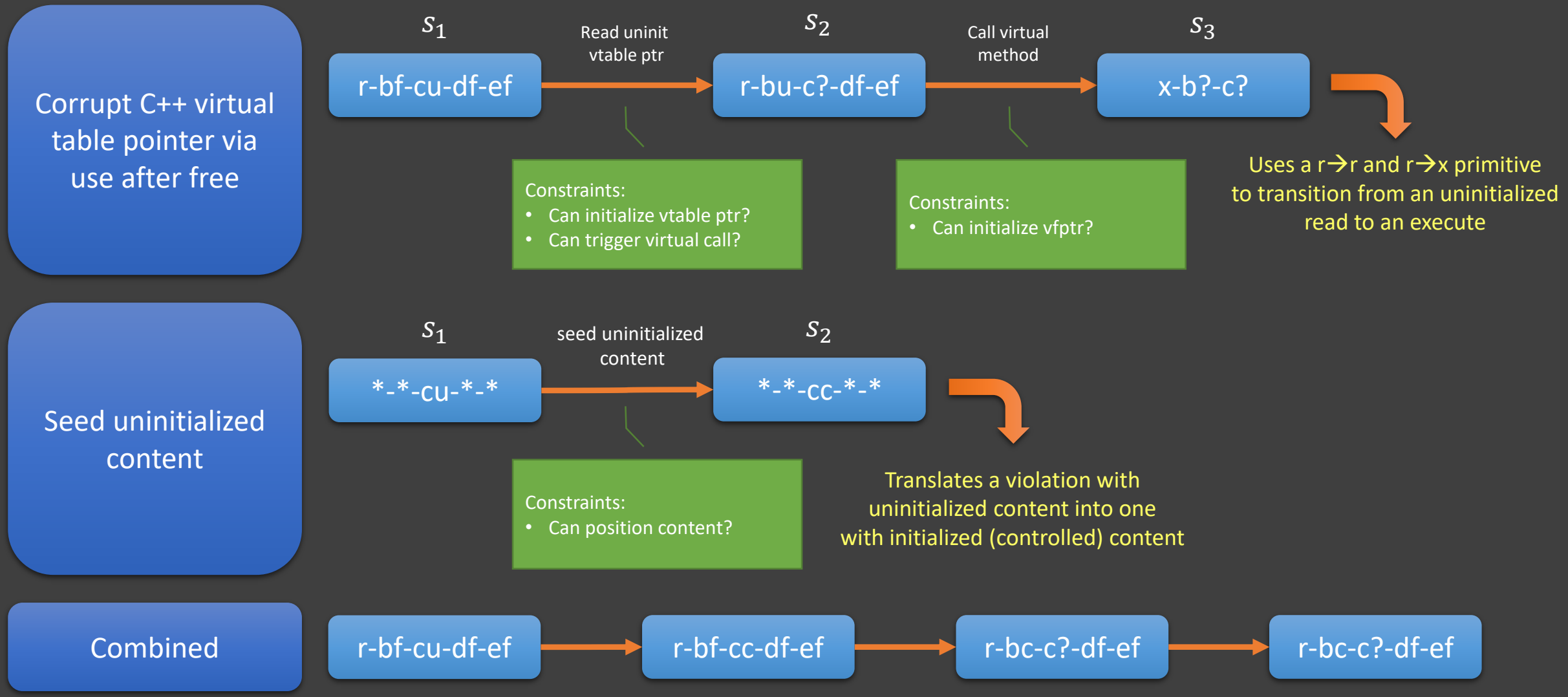
Stack return address overwrite



Convert a relative write into an absolute write

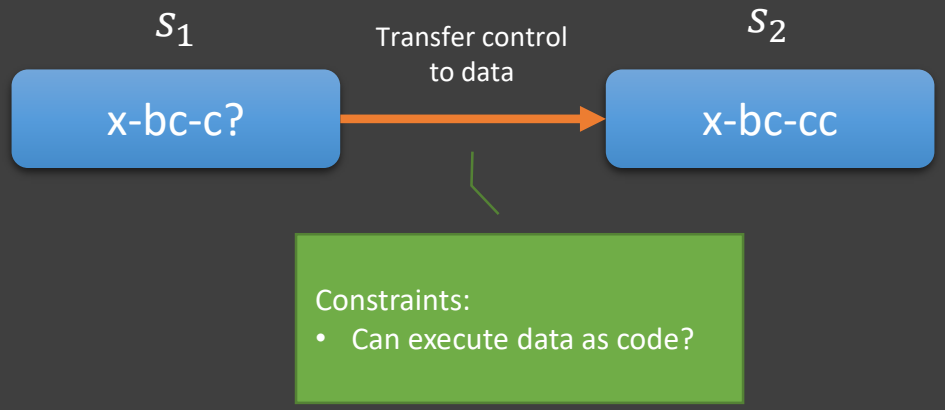


Examples of exploitation techniques (II)

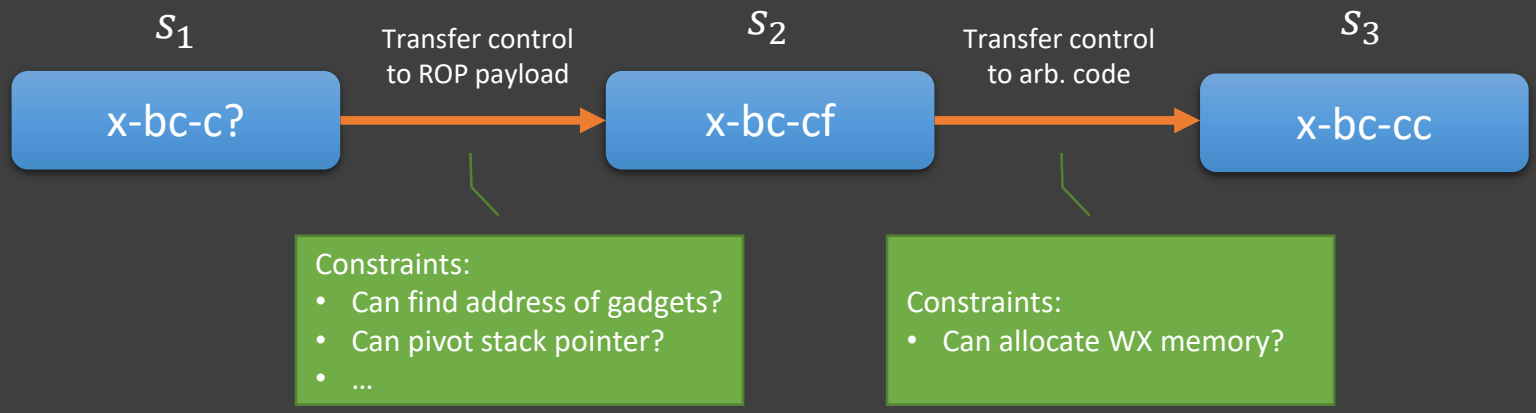


Examples of exploitation techniques (III)

Execute data as code



Execute ROP stage to arbitrary code



Chaining exploitation techniques

Exploitation techniques can be chained together to reach a desired end state

Example #1: stack buffer overrun

From violation	Transition	To violation
w-bf-cc-df-ec	corrupt ret addr	r-b?-cc-d?-e?
r-b?-cc-d?-e?	return from func	x-bc-c?
x-bc-c?	load non-ASLR img	x-bc-cf
x-bc-cf	execute ROP stage	x-bc-cc

Example #2: C++ object use after free

From violation	Transition	To violation
r-bf-cu-df-ef	init vtable ptr	r-bf-cc-df-ef
r-bf-cc-df-ef	read vtable ptr	r-bc-c?-d?-e?
r-bc-c?-d?-e?	spray vtable	r-bc-cc-d?-e?
r-bc-cc-d?-e?	call virt method	x-bc-c?
x-bc-c?	load non-ASLR img	x-bc-cf
x-bc-cf	execute ROP stage	x-bc-cc

Exploitability of both chains depends on probability of satisfying the constraints of each transition

Simulation Demo

Applying this model

Measuring value, guiding investments, and improving risk assessment

Measuring the value of defensive technologies

No established system exists to measure the impact of defensive technologies like as DEP, ASLR, and /GS

Impact is often measured based on the ability to break techniques used by public exploits[18]

E.g. “exploit X would not have worked with mitigation Y in place”

- Impact is measured based on past behavior
- Absence of relevant exploits can be problematic
  - Hard to justify new tech without data
- Does not quantify difficulty of exploitation *after* enabling mitigation
  - Impact measurement is hindsight only

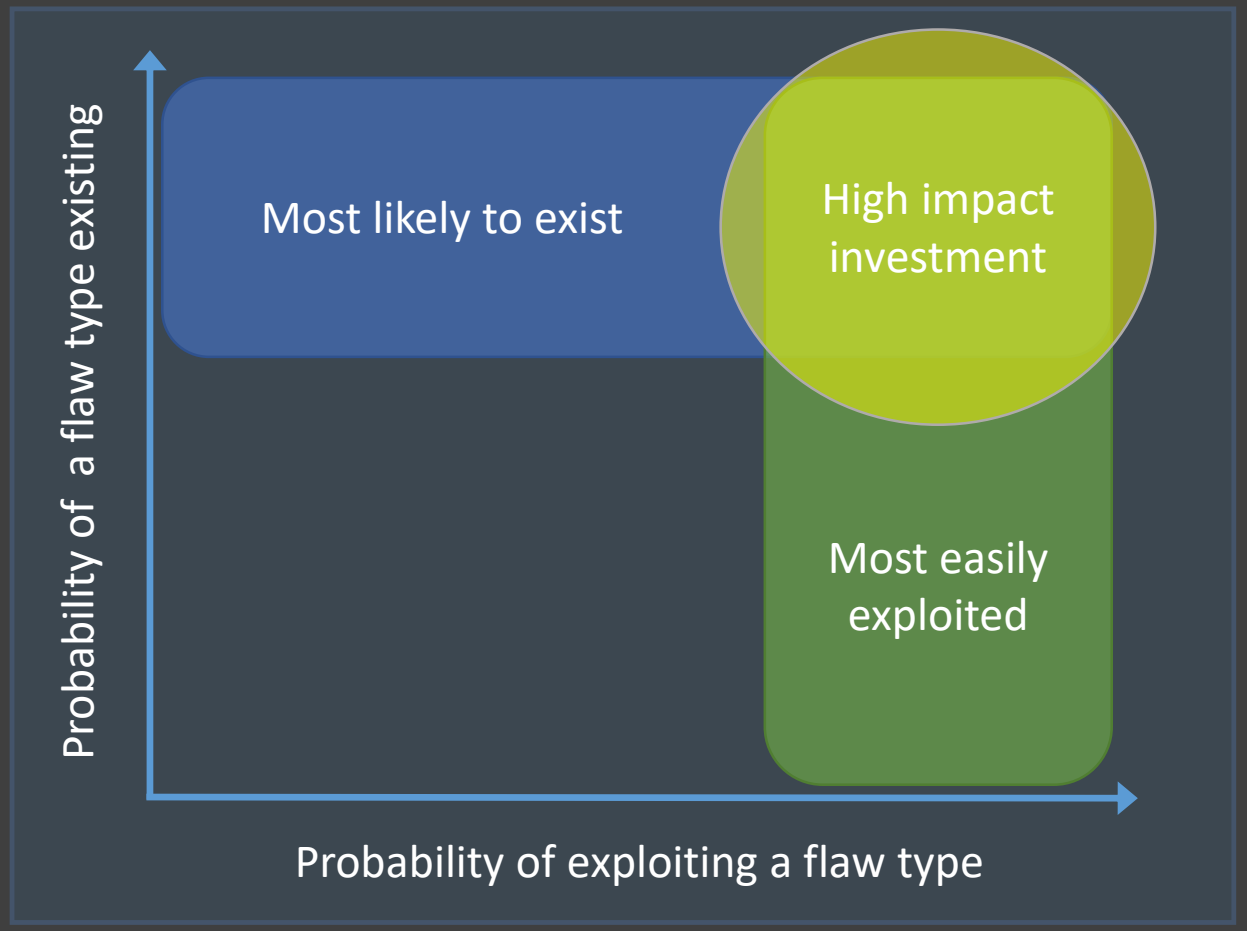
A model of exploitation can be used to concretely measure the impact of defensive technologies

E.g. “with mitigation Y in place, the exploitability in a given scenario is Z”

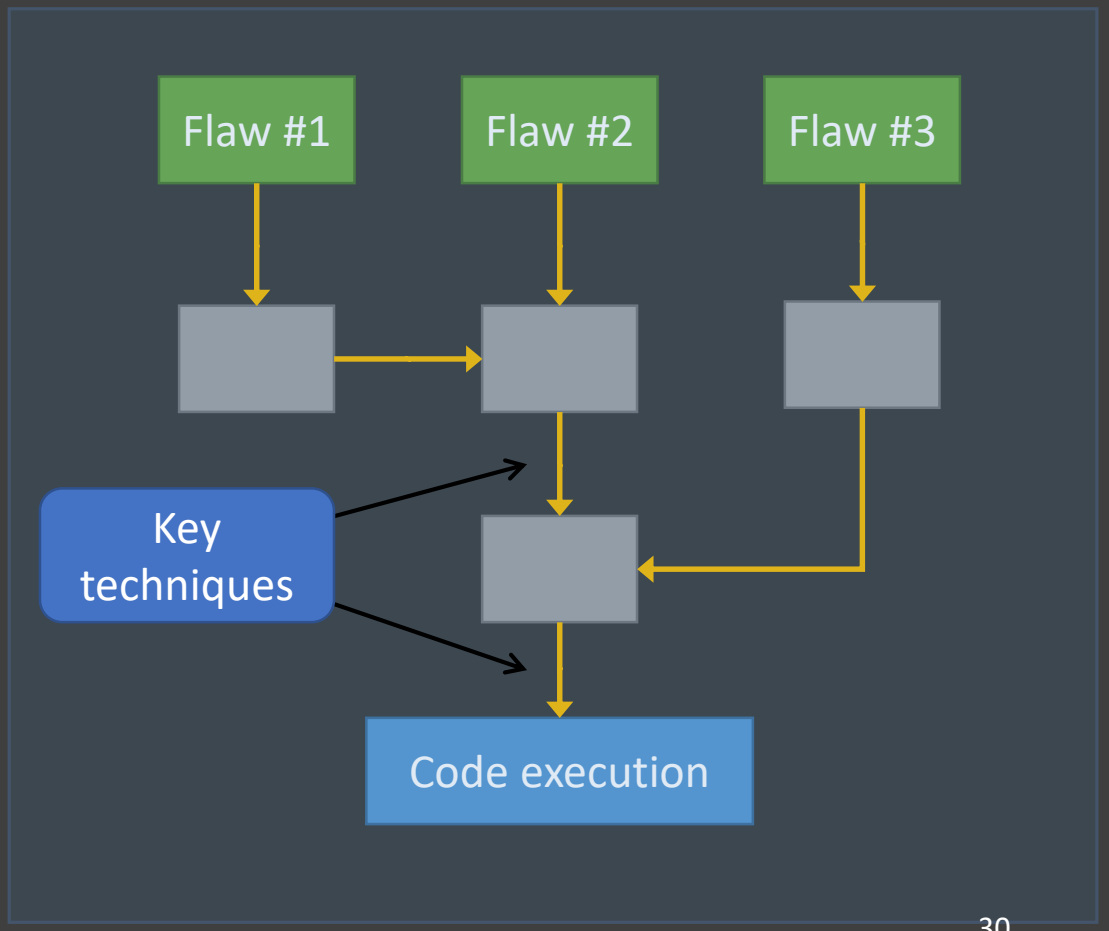
- Impact is measured based on possible behavior
- Open to scrutiny, refinement, and customization
  - Impact may change because of new exploitation techniques
- Facilitates more rigorous “what-if” scenarios
  - E.g. exploring value of a new mitigation

Guiding defensive investments

Invest in detection & mitigation of flaw types that are most easily exploited & most likely to exist



Invest in mitigations for exploitation techniques that are key to exploiting many types of flaws



Improving risk management

A model of exploitation could enable more granular and effective risk management

Provide a measure of exploitability for each affected product version rather than generalizing

Enable level of acceptable risk to be customized based on assumptions about an attacker's abilities

Enable continuous and transparent refinement based on changes in the state of the art

Provide an objective description of exploitability that can be more easily agreed upon or refuted

Conclusion & next steps

Conclusion

- Measuring risk associated with memory safety vulnerabilities is challenging
  - Often requires conservative analysis which may not reflect actual risk
- An abstract model of memory safety could help improve on this situation
  - Provide a measure of exploitability that enables more effective risk mgmt

Next steps

- Feedback: this is where you come in ☺
  - Any and all feedback welcome, especially critical feedback!
  - Have something that would be hard to model? I want to hear about it!
- Continue to refine the current prototype of the model described today

Questions?

Microsoft®

Be what's next.™

References

[1] Sean Heelan. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. PhD thesis. Sep, 2009. http://seanhn.files.wordpress.com/2009/09/thesis1.pdf.

[2] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. *Automatic Patch-Based Exploit Generation.* In Proceedings of the IEEE Symposium on Security and Privacy, May 2008. http://www.cs.cmu.edu/~dbrumley/pubs/apeg.pdf.

[3] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, David Brumley. *AEG: Automatic Exploit Generation*. Network and Distributed System Security Symposium, Feb 2011. http://security.ece.cmu.edu/aeg/aeg-ndss-2011.pdf.

[4] Microsoft Security Engineering Center (MSEC). *!exploitable Crash Analyzer*. Jun, 2009. http://msecdbg.codeplex.com/.

[5] Apple Product Security. *Announcing CrashWrangler*. Jul, 2009. http://seclists.org/dailydave/2009/q3/11.

[6] Microsoft. *Microsoft Exploitability Index*. Oct, 2008. http://technet.microsoft.com/en-us/security/cc998259.aspx.

[7] MITRE. *Common Weakness Enumeration (CWE)*. http://cwe.mitre.org/data/index.html.

[8] Sandeep Bhatkar, Eep Bhatkar, R. Sekar, Daniel C. Duvarney. *Efficient Techniques for Comprehensive Protection from Memory Error Exploits*. USENIX Security, 2005. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.9963.

[9] Matt Miller. *State of the Exploit*. ToorCon Seattle, 2008. http://hick.org/~mmiller/presentations/seatoorcon08/2008_seatoorcon.ppt.

[10] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, Anna Shubina. *Exploit Programming*. 2011. http://www.cs.dartmouth.edu/~sergey/langsec/papers/Bratus.pdf.

[11] Chris Valasek and Ryan Smith. *Exploitation in the Modern Era*. Black Hat Europe, 2011.

[12] Halvar Flake. *Exploitation and state machines*. Infiltrate, 2011. http://immunityinc.com/infiltrate/archives/Fundamentals_of_exploitation_revisited.pdf.

[13] Patroklos Argyroudis, Chariton Karamitas. *Heap Exploitation Abstraction by Example*. OWSAP AppSecResearch, 2012. http://census-labs.com/media/heap-owasp-appsec-2012.pdf.

[14] Microsoft. *Microsoft Security Intelligence Report: Volume 7*. 2009. http://download.microsoft.com/download/A/3/0/A30A60D9-1303-4B6A-91B7-BB24E0211B05/Microsoft_Security_Intelligence_Report_volume_7_Jan-Jun2009.pdf.

[15] Adel Abouchaev, Damian Hasse, Scott Lambert, Greg Wroblewski. *Analyze Crashes to Find Security Vulnerabilities in Your Apps*. MSDN Magazine, Nov, 2007. http://msdn.microsoft.com/en-us/magazine/cc163311.aspx.

References

[16] Edward J. Schwartz, Thanassis Avgerinos, David Brumley. *Q: Exploit Hardening Made Easy*.

In the Proceedings of the 2011 USENIX Security Symposium, August, 2011. http://www.ece.cmu.edu/~ejschwar/papers/usenix11.pdf

[17] Jonathan D. Pincus, Brandon Baker. *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*. IEEE Security & Privacy, vol 2, 2004. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1324594.

[18] Dan Guido. *Exploit Intelligence.* SOURCE Boston, 2011. http://www.isecpartners.com/storage/docs/presentations/EIP-final.pdf.