Vulnerability Reachability Analysis Using OSS Tools

Mike Larkin
Deepfactor, Inc.
mlarkin@deepfactor.io

# # #

{df}

// Agenda

> Overview (~20 minutes)

> Types of reachability analysis (~10 minutes)

> Call graph analysis exercise (~10 minutes)

> Dynamic/runtime analysis exercise (~10 minutes)

> Results comparison (~10 minutes)

> Conclusion / Q&A (~10 minutes)

Overview

# # #

{df}

// About Me

- > Co-founder and CTO at Deepfactor
  - We make software to help people prioritize vulnerability remediation
- > Adjunct faculty at San Jose State University
  - Computer Engineering (CMPE) MS degree program
  - Virtualization Technologies, Software Security, and Operating Systems
- > Active open-source contributor
  - OpenBSD (hypervisor, device drivers, memory/device management, ACPI)

//Goals Of This Workshop

> This workshop has several goals; at the end of the workshop, you should -

- Know what reachability analysis is, and why you should care about it
- Know why reachability can help you prioritize vulnerability remediation
- Understand the different types of reachability analysis tools
- Learn where you can reach out for help in this area later

// If You Want To Follow Along …

> I will be doing 3 examples today that you can also do yourself

- … if you want. Otherwise, sit back and relax and enjoy the beer and food

> The list of what you will need to install is pretty simple:

- Trivy https://trivy.dev
- Go https://go.dev
- Java https://openjdk.org
- Gradle https://gradle.org (if you want to try the Java example)

> For the Go example, you'll need some Go app (of your choice)

- I'm going to demo JIRA-CLI : https://github.com/ankitpokhrel/jira-cli

{df}

//Vulnerability Reachability Analysis

> Code that is contains vulnerabilities is bad

> Code that contains vulnerabilities *used in your application* is worse

> How do you know if some code you are using is vulnerable?

> Better yet, how do you **know** you're *even using* the vulnerable code at all?

> These questions are what we are going to focus on today

//Vulnerability Reachability Analysis

> We will start by talking about reachability

> We'll then talk about what vulnerabilities are, and how they are managed

> Then we will look at tools you can use to catalog what CVEs you might have in your code

> Finally, we'll conclude with some short examples with open source tools to do your own reachability analysis

Reachability

# # #

{df}

//Reachable Code

- > How do you define reachability?
- > Certainly, code that your program executes is, by definition, reachable
- > What about code that is packaged with your program but never loaded?
- > What about code that is loaded by your program but never executed?
- > What about code sitting on the same machine/container that could theoretically be launched?

//Reachable Code

> “Code that is packaged with your program but never loaded”

> I’d suggest getting rid of that code

> There are tools to help you locate such code

// Reachable Code

> “Code that is loaded by your program but never executed”

> For example
- Shared library dependencies created by the linker but not used
- Java apps doing Class.forName(…) but never using any methods in the class
- dlopen(…) but never using the thing you loaded

> This might happen in applications that support things like plugins, but then the loaded module isn’t ever exercised

> Code like this is reachable!

//Reachable Code

> “Code sitting on the same machine that might be launched”

> Out of scope (for this talk…)

> This is sort of like the earlier example though; if it’s not used, why is it there?

> No need to leave lolbins laying around for an attacker

{df}

//Reachable Code

> If we distill the previous scenarios down to the two important ones …
- Code directly executed by your program
- Code loaded into the address space/interpreter by your program (maybe used, maybe not)

> How do you know which functions/methods fall into each category?

> Said a different way, how can you compile a definitive list of functions and methods that are reachable, according to the previous definitions?

//Reachability Analysis

> Before we discuss "how", let's talk about "why"

> Why is creating this list important?

> Simple answer –

Reachable code that contains vulnerabilities should be remediated with priority

//Reachability Analysis

> If you have several vulnerabilities to fix…

- Prioritize fixing the ones that are reachable, with known exploit PoCs first
- Next focus on the other reachable ones
- Then focus on the rest, based on severity

> All that advice depends on *knowing what is reachable*

//Reachability Analysis

> There are generally two types of reachability analysis tools

- Tools that scan *source code* and generate a *call graph* based on *syntax analysis*
  > foo().bar().baz()   -> "methods foo, bar, baz are reachable"

- Tools that monitor the program after it is built, and watch what is loaded or executed
  > Profiling, library call interception, etc

> Each of these approaches can produce a list of reachable functions/methods

> Each approach has strengths and weaknesses

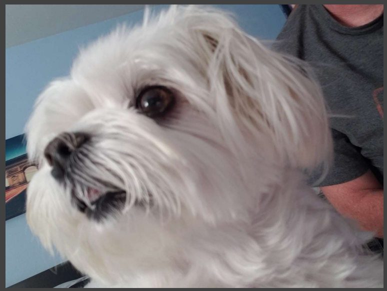
Vulnerabilities & Bad Code

# # #

{df}

//Let's Talk About Software Vulnerabilities

> Vulnerable code is everywhere.
- You're using it
- I'm using it
- Even my dog is using it



> Let's talk about where vulnerable code comes from

//Let's Talk About Software Vulnerabilities

> What causes a vulnerability?

- Are vulnerabilities caused by incorrect (buggy) code?
- Is correct code vulnerability free?
- Is vulnerability free code always correct?
- Are vulnerabilities in your program always the result of code ***you*** wrote?

// Stupid Example

What do we think about this?

Is this correct (bug free)?

Could this code have a vulnerability?

```
int data[MAX_DATA];

/* Return data at position "index" */
int
function(int index)
{
        int i;

        i = data[index];

        return i;
}
```

// Stupid Example #2

What do we think about this?

Is this correct (bug free)?

Could this code have a vulnerability?

```
int data[MAX_DATA];

/* Return data at position "index" */
int
function(int index)
{
	int i = -1;

	if (index < MAX_DATA)
		i = data[index];

	return i;
}
```

{df}

// Stupid Example #3

What do we think about this?

Is this correct (bug free)?

Could this code have a vulnerability?

```
int *numbers;

/*
 * set numbers[x] = x for all x > 0 and < size .
 */
void
function(int size)
{
        int i;

        numbers = malloc(size * sizeof(int));

        for (i = 0; i < size; i++)
                numbers[i] = i;
}
```


// Yes, Those Were Stupid

> Type confusion
  - Misunderstanding the meaning of a value

> Corner cases
  - Not checking for all error conditions

> Not checking return values

> Undefined behavior

  - Of course, nobody here would ever make such mistakes…

//Not So Obvious Example

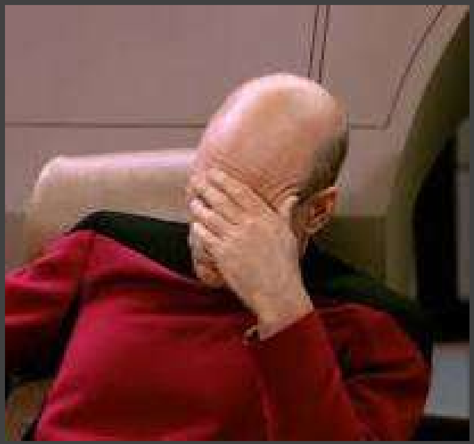
```
static unsigned int
tun_chr_poll(struct file *file, poll_table * wait)
{
        struct tun_file *tfile = file->private_data;
        struct tun_struct *tun = __tun_get(tfile);
        struct sock *sk = tun->sk;
        unsigned int mask = 0;

        if (!tun)
                return POLLERR;

        . . .
}
```

https://lwn.net/Articles/342330

//Not So Obvious Example



https://lwn.net/Articles/342330

```
static unsigned int
tun_chr_poll(struct file *file, poll_table * wait)
{
        struct tun_file *tfile = file->private_data;
        struct tun_struct *tun = __tun_get(tfile);
        struct sock *sk = tun->sk;
        unsigned int mask = 0;

        if (!tun)
                return POLLERR;

        . . .
}
```

{df}

//Why Did We Look At These Stupid Examples?

> These examples were shown to illustrate a few points

- Even simple mistakes or accidents can cause a vulnerability
- Vulnerabilities are everywhere
- They are not going away
- You probably didn’t write the bad code yourself
- We need a way to track them and prioritize remediation

//Bad Code Is Out There

> We'll never be able to get rid of bad code

> Mistakes, laziness, apathy, and inexperience can all contribute to the problem

- (Ehm, memory unsafe languages, too)

> Even if you write 100% perfect bug-free, vulnerability-free code, you are still likely to step on landmines

- Importing third party code/dependencies
- Downstream refactoring
- Code being used in unexpected ways

// CVEs

> Known vulnerabilities can be assigned a CVE number for tracking
  - Each CVE is assigned a severity
  - Each CVE can contain information about the vulnerability
  - Each CVE can contain information about "fixed-in" versions
  - … plus arbitrarily more information …

> Who assigns CVEs?

> What are they used for?

> Who decides the severity and other information included in the report?

{df}

//Example Of A Meaningless CVE

Vulnerability Details : CVE-2021-41340

Windows Graphics Component Remote Code Execution Vulnerability

Published 2021-10-13 01:15:13 Updated 2023-08-01 23:15:24 Source Microsoft Corporation

Vulnerability category: Execute code

Exploit prediction scoring system (EPSS) score for CVE-2021-41340

Probability of exploitation activity in the next 30 days: 27.53%

Percentile, the proportion of vulnerabilities that are scored at or less: ~ 97 % EPSS Score History EPSS FAQ

CVSS scores for CVE-2021-41340

Base Score	Base Severity	CVSS Vector	Exploitability Score	Impact Score
6.8	MEDIUM	AV:N/AC:M/Au:N/C:P/I:P/A:P	8.6	6.4
7.8	HIGH	CVSS:3.1/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H	1.8	5.9
7.8	HIGH	CVSS:3.1/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H	1.8	5.9

// Better Example

Vulnerability Details : CVE-2023-32235

Ghost before 5.42.1 allows remote attackers to read arbitrary files within the active theme's folder via /assets/built%2F..%2F..%2F/ directory traversal. This occurs in frontend/web/middleware/static-theme.js.

Published 2023-05-05 05:15:09 Updated 2023-05-11 14:19:32 Source MITRE

View at NVD, CVE.org

Vulnerability category: Directory traversal

Exploit prediction scoring system (EPSS) score for CVE-2023-32235

Probability of exploitation activity in the next 30 days: 89.91%

Percentile, the proportion of vulnerabilities that are scored at or less: ~ 99 %

EPSS Score History EPSS FAQ

// CVEs (cont'd)

> How do you know if you're vulnerable to a CVE?

> To answer the question, it's important to know ***what components*** you are using in your application

- After all, if you aren't using component XYZ at all, then you're certain to not be subject to any of its vulnerabilities

> Ok, so how do you know what components you are using in your application?

{df}

// Imports

> If you're lucky, your language or compiler might tell you
  - For example, go.mod

> The developer might also tell you
  - Gradle or .pom files
  - package_lock.json

> Or maybe you can scan your program and try determine what it uses, if you don't know

```
require (
  github.com/AlecAivazis/survey/v2 v2.3.7
  github.com/atotto/clipboard v0.1.4
  github.com/briandowns/spinner v1.23.0
  github.com/charmbracelet/glamour v0.6.0
  github.com/cli/safeexec v1.0.1
  github.com/fatih/color v1.15.0
  github.com/gdamore/tcell/v2 v2.6.0
  github.com/google/shlex v0.0.0-20191202
  github.com/kballard/go-shellquote v0.0.0-201851
  github.com/kentaro-m/blackfriday-confluence v0.0.0-2022
  github.com/kr/text v0.2.0
  github.com/mattn/go-isatty v0.0.19
          . . .
  github.com/alecthomas/chroma v0.10.0 // indirect
  github.com/alessio/shellescape v1.4.1 // indirect
  github.com/aymanbagabas/go-osc52/v2 v2.0.1 // indirect
  github.com/aymerick/douceur v0.2.0 // indirect
  github.com/cpuguy83/go-md2man/v2 v2.0.2 // indirect
  github.com/creack/pty v1.1.18 // indirect
  github.com/danieljoos/wincred v1.2.0 // indirect
  github.com/davecgh/go-spew v1.1.1 // indirect
  github.com/dlclark/regexp2 v1.10.0 // indirect
  github.com/fsnotify/fsnotify v1.6.0 // indirect
          . . .
)
```

//Example - Trivy

> Trivy can be used to scan a program’s dependencies
  - Plus container images, filesystems, etc

  - https://github.com/aquasecurity/trivy

> Let’s scan a container

// SBOMs

> Software Bill Of Materials

- Similar to a BOM for a physical thing like a car, toaster, or television
- Lists all the things required to build the “thing” (software in this case)
  > Instead of nuts, bolts, flanges, and circuit boards, we have lists of software packages and their versions
- Can be described in various formats (SPDX, CycloneDX)

> Biden executive order 14028

- https://www.ntia.gov/sites/default/files/publications/sbom_myths_vs_facts_nov2021_0.pdf

{df}

// SBOMs (cont'd)

> With an SBOM, an organization is empowered to …

- Answer the question "Am I affected?" more easily when a vulnerability is discovered
  > Minutes or hours, not days or weeks later
- Determine which components are affected
- Determine roadmaps for remediation, when coupled with *reachability insights*

// SBOMs (cont'd)

> SBOM content can be *correlated* with CVE databases

> This would give you a list containing two things

- Components used to build your application
- Vulnerabilities present in those components

> Surely that be enough to prioritize what gets fixed first, right?

// Sample SBOM

libexpat1	0.57%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.2.6-2+deb10u4
libldap-2.4-2	1.1%	balancereader:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.4.57+dfsg-3
libtinfo6	0.1%	transactionhistory:0...	public.ecr.aws/dee...	OS Package	debian	6.2+20201114-2
python2.7-minimal	4.04%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.7.16-2+deb10u1
libss2	0.07%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	1.44.5-1+deb10u3
libcurl4-openssl-dev	17.09%	fioapp:0.0.2	public.ecr.aws/dee...	OS Package	ubuntu	7.58.0-2ubuntu3
libidn2-0	0.35%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.0.5-1+deb10u1
org.springframework.boot:spring-boot	0.04%	transactionhistory:0...	public.ecr.aws/dee...	Dependency	jar	2.3.1.RELEASE
libssh2-1	0.05%	transactionhistory:0...	public.ecr.aws/dee...	OS Package	debian	1.9.0-2
libc6	2.15%	transactionhistory:0...	public.ecr.aws/dee...	OS Package	debian	2.31-13+deb11u2
libwind0-heimdal	1.37%	userservice:0.0.1	public.ecr.aws/dee...	OS Package	ubuntu	7.5.0+dfsg-1
org.apache.httpcomponents:httpclient	0.16%	transactionhistory:0...	public.ecr.aws/dee...	Dependency	jar	4.5.12

20 rows ▾ 1–20 of 784

{df}

// Sample SBOM

libexpat1	0.57%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.2.6-2+deb10u4
libldap-2.4-2	1.1%	balancereader:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.4.57+dfsg-3
libtinfo6	0.1%	transactionhistory:0...	public.ecr.aws/dee...	OS Package	debian	6.2+20201114-2
python2.7-minimal	4.04%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.7.16-2+deb10u1
libss2	0.07%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	1.44.5-1+deb10u3
libcurl4-openssl-dev	17.09%	fioapp:0.0.2	public.ecr.aws/dee...	OS Package	ubuntu	7.58.0-2ubuntu3
libidn2-0	0.35%	ninja-js:0.0.1	public.ecr.aws/dee...	OS Package	debian	2.0.5-1+deb10u1
org.springframework.boot:spring-boot	0.04%	transactionhistory:0...	public.ecr.aws/dee...	Dependency	jar	2.3.1.RELEASE
libssh2-1	0.05%	transactionhistory:0...	public.ecr.aws/dee...	OS Pa		1.9.0-2
libc6	2.15%	transactionhistory:0...	public.ecr.aws/dee...	OS Pa		2.31-13+deb11u2
libwind0-heimdal	1.37%	userservice:0.0.1	public.ecr.aws/dee...	OS Pa		7.5.0+dfsg-1
org.apache.httpcomponents:httpclient	0.16%	transactionhistory:0...	public.ecr.aws/dee...	Depen		4.5.12

20 rows

1–20 of 784

Yikes!

784 Vulnerable Components?!?

// Tidal Wave

> That's not solvable

> You're going to get crushed by the neverending wave of CVEs

> Let's fix the problem

Call Graph Reachability Analysis

# # #

{df}

//Call Graphs

> There are tools that can tell you what code is reachable in your application at build time

> These tools scan your source code and produce a graph of “what calls what”

> That graph is then traversed to create a list of reachable code paths

> The hope here is that by knowing what is *possibly callable,* we can define the list of reachable code

> With that information in hand, we should be able to prioritize remediation tasks

// Call Graphs

> A compiler analyzes your code during build and creates a syntax tree

> Some nodes in this tree can be call sites (locations where program flow transitions from one function to another)

> Call sites can be cataloged to create the “what calls what” list

```
foo()
{
        bar();
}

bar()
{
        baz();
}

baz()
{
        . . .
}
```

// Call Graphs

> In this example, we know that foo calls bar and bar calls baz

> Assuming foo is called from somewhere else, then our list of reachable code consists of

- foo
- bar
- baz

```
foo()
{
        bar();
}

bar()
{
        baz();
}

baz()
{
        . . .
}
```

{df}

//Call Graphs

> This list can help us prioritize remediating any CVE that includes one of these functions

- Eg, "A remote code execution vulnerability exists in libFooBarBaz.so if the baz() function is called."

```
foo()
{
        bar();
}

bar()
{
        baz();
}

baz()
{
        . . .
}
```

//Call Graphs

> A different example

> What can we say about the reachability of "hamburger"?

> It's not called from anywhere

> Is it reachable?

```
foo()
{
        bar();
}

bar()
{
        printf("hello");
}

hamburger()
{
        . . .
}
```

//Call Graphs

> Language complexities make it difficult to catch all the cases
- Function pointers
- Reflection based invocation
- Function names not known at compile time

> Is bar() called here? What about baz()?

> Are either of them or both reachable?

```
foo()
{
        if (some_param) == 42
                ptr = baz;
        else
                ptr = bar;

        ptr();
}

bar()
{
        . . .
}

baz()
{
        . . .
}
```

// Call Graphs

> What can we say about the reachability of various code here?

> Is any code even executed from class "name" in this example?

> It's difficult to get a complete picture of what's going if all you have to look at is the source

```
public void myMethod(String name)
{
        Object o =
          class.forName(name);

        . . .
}
```

//Call Graph Example

> I'll be showing how to produce a call graph from a Go application using 'callgraph'

- https://github.com/golang/tools/tree/master

> This tool should work against any Go application for which you have source

Runtime Reachability Analysis

# # #

{df}

//Runtime Reachability

> Tools that use *runtime reachability analysis* create the list of reachable code by examining the program while it runs

> These tools generally do not look at source code, although they may, for additional context (eg, if the tool also produces SBOMs)

> By monitoring what is used by the program, the list of reachable code can be created

{df}

//Runtime Reachability

> Since the list of reachable code is defined by what is used during monitoring, care must be taken to ensure the system under test is exercised fully

> Tools employing runtime reachability can have different granularities

- Function level
- Module level

> Function level tracing gives more specificity but can produce substantial output

> Module level tracing omits some specificity and assumes “module loaded” means “code in that module is reachable”

// Runtime Reachability

> How do these tools work?

> Some intercept library calls to monitor when specific functions are called

> Some use traditional profiling techniques (periodic stack sampling)

> Some emulate or partially emulate the program’s execution to monitor calls

> Each approach is slightly different but all fall under the category of runtime analysis

{df}

// Runtime Reachability Example

> I'll be showing how to generate a list of called/used Java classes at runtime using a small bytecode rewriting agent

- The agent can be found here:
- https://github.com/deepfactor-io/reachability-workshop

Putting It All Together

Reachability + Prioitization

# # #

{df}

// Recap

> Ok, so at this point we've done the following

- Scanned our application and produced an SBOM
- Using the SBOM, correlated which CVEs we *might* be vulnerable to based on the SBOM contents
- Performed a reachability analysis exercise on our code which gave us a list of modules or functions used

> How do we put all this together to arrive at a prioritization order?

{df}

// First Step – Code-To-Module

> Let's pretend that we have built a list of reachable code that looks like this
  - The list could have been created using either approach (call graph analysis or runtime analysis)

> What's next?

```
/usr/lib/libfoo.so
        foo()
        bar()
        baz()

/usr/lib/libyummy.so
        hamburger()
        hotdog()
        sushi()

/usr/bin/myapp
        main()
        func1()
        func2()
```

{df}

// First Step – Code-To-Module

> We need to get from this list of modules to something that matches what we have in our SBOM

> Remember, CVE lists are often sourced from the software vendor and thus will be using vendor package names
  - Eg, “libyummy-1.2.3p1” not “/usr/lib/libyummy.so”

```
/usr/lib/libfoo.so
        foo()
        bar()
        baz()

/usr/lib/libyummy.so
        hamburger()
        hotdog()
        sushi()

/usr/bin/myapp
        main()
        func1()
        func2()
```

// First Step – Code-To-Module

> Assuming you have a package manager, the reverse file mapping capability is useful here

- rpm –qf
- dpkg –S
- apk info --who-owns
- . . .

> P.S. This is one reason a package manager is important …

```
/usr/lib/libfoo.so
        foo()
        bar()
        baz()

/usr/lib/libyummy.so
        hamburger()
        hotdog()
        sushi()

/usr/bin/myapp
        main()
        func1()
        func2()
```

{df}

// Second Step – Module-To-CVE

> Now that we have the list of modules, a query against the list of CVEs we obtained previously can be made
  - grep, sed, awk, jq, whatever…
  - Can add thresholds or ordering in this step, based on your organization's appsec policies

> Note: Your own executable/class probably won't be packaged this way
  - And even if it was, it would be you issuing CVEs for it anyway

```
/usr/lib/libfoo.so :: libfoo-61.7
/usr/lib/libyummy.so :: libyummy-1.3
/usr/bin/myapp


$ jq '.[package]' cvelist.json | grep libfoo

CVE-2024-12345:
 A vulnerability exists in libfoo's baz()
 function …
```

{df}

// Second Step – Module-To-CVE

- Sometimes the CVE text will tell you definitively which function is bad
- Most of the time you need to be content with just assuming if you used *anything* in the module that you should throw it out or upgrade
  - Vendors are disincentivized to provide real useful information

```
/usr/lib/libfoo.so :: libfoo-61.7
/usr/lib/libyummy.so :: libyummy-1.3
/usr/bin/myapp


$ jq '.[package]' cvelist.json | grep libfoo

CVE-2024-12345:
 A vulnerability exists in libfoo's baz()
 function …
```

{df}

// Second Step – Module-To-CVE

> In the end, we've produced the following

- A list of CVEs …
- … applicable to modules we have in our SBOM
- … that we *provably used code from* in our program

> Using this approach, we now have a list of the "most important" CVEs

```
/usr/lib/libfoo.so :: libfoo-61.7
/usr/lib/libyummy.so :: libyummy-1.3
/usr/bin/myapp


$ jq '.[package]' cvelist.json | grep libfoo

CVE-2024-12345:
 A vulnerability exists in libfoo's baz()
 function …
```

{df}

// Final Step

> Of course, you could take this further

- Further refine the list to prioritize CVEs with known public exploits

> EPSS score is a way of tracking this

- "Is an exploit available?"
- "What is the likelihood of an exploit *becoming* available in the next 30/60/90 days?
- Some tools incorporate EPSS into their severity ranking

> VEX enhancements to CVEs

- Sometimes more information can be gleaned

Conclusion

# # #

{df}

// Conclusion

> What have we learned?

- We learned what it means for code to be reachable
- We learned why we need to care about vulnerabilities
- We learned how to scan our code for CVEs
- We learned how to apply reachability analysis to discover which modules have reachable code
- We learned how to create a prioritized list of CVEs based on reachability

{deepfactor}

Thank You