Why Johnny and Janie Can't Code Safely: Bringing Software Assurance to the Masses (and a ramble through other topics)

USENIX Security ◆ August 13, 2015

Barton P. Miller
Chief Scientist, DHS Software Assurance Marketplace
Professor, Computer Sciences
University of Wisconsin

The obligatory

FUD

(fear, uncertainty, doubt)

slide …

A Jaw-Dropping Increase in Software…

150+ million websites

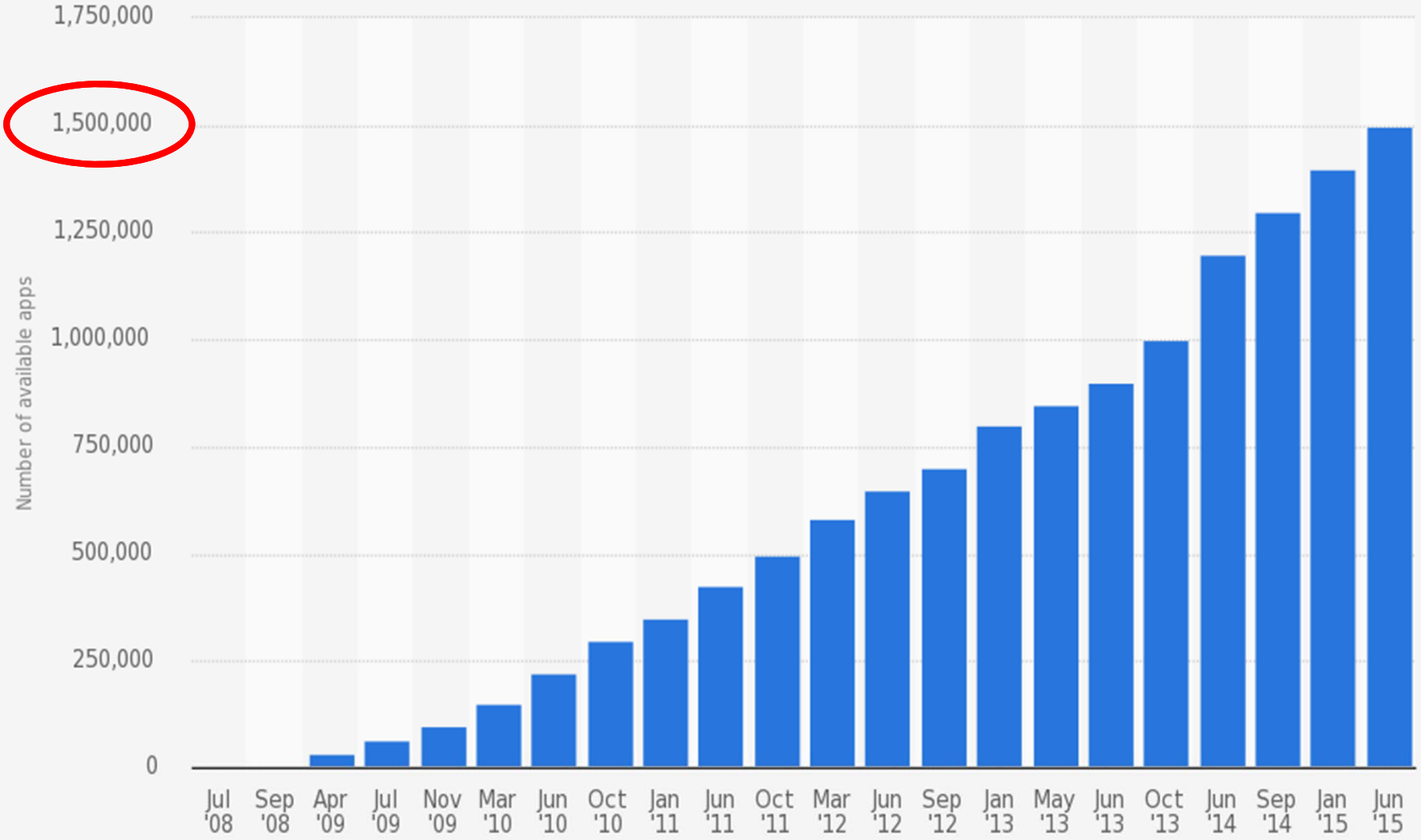
Open source software

Internet of *things*

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

CONTINUOUS ASSURANCE

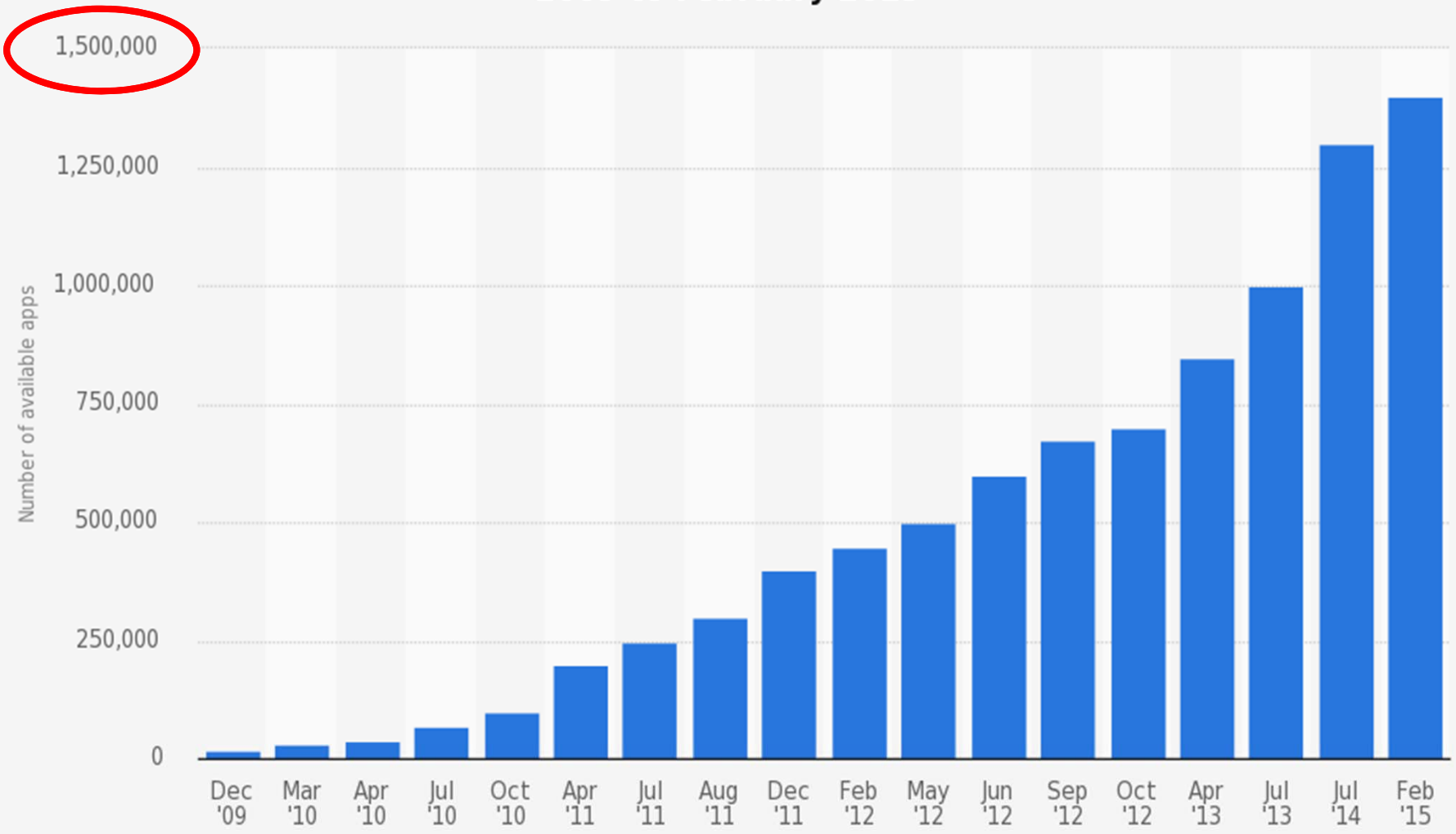
Number of available apps in the Apple App Store from July 2008 to June 2015



Source::
Apple

Additional Information:
Worldwide; July 2008 to June 2015

Number of available applications in the Google Play Store from December 2009 to February 2015

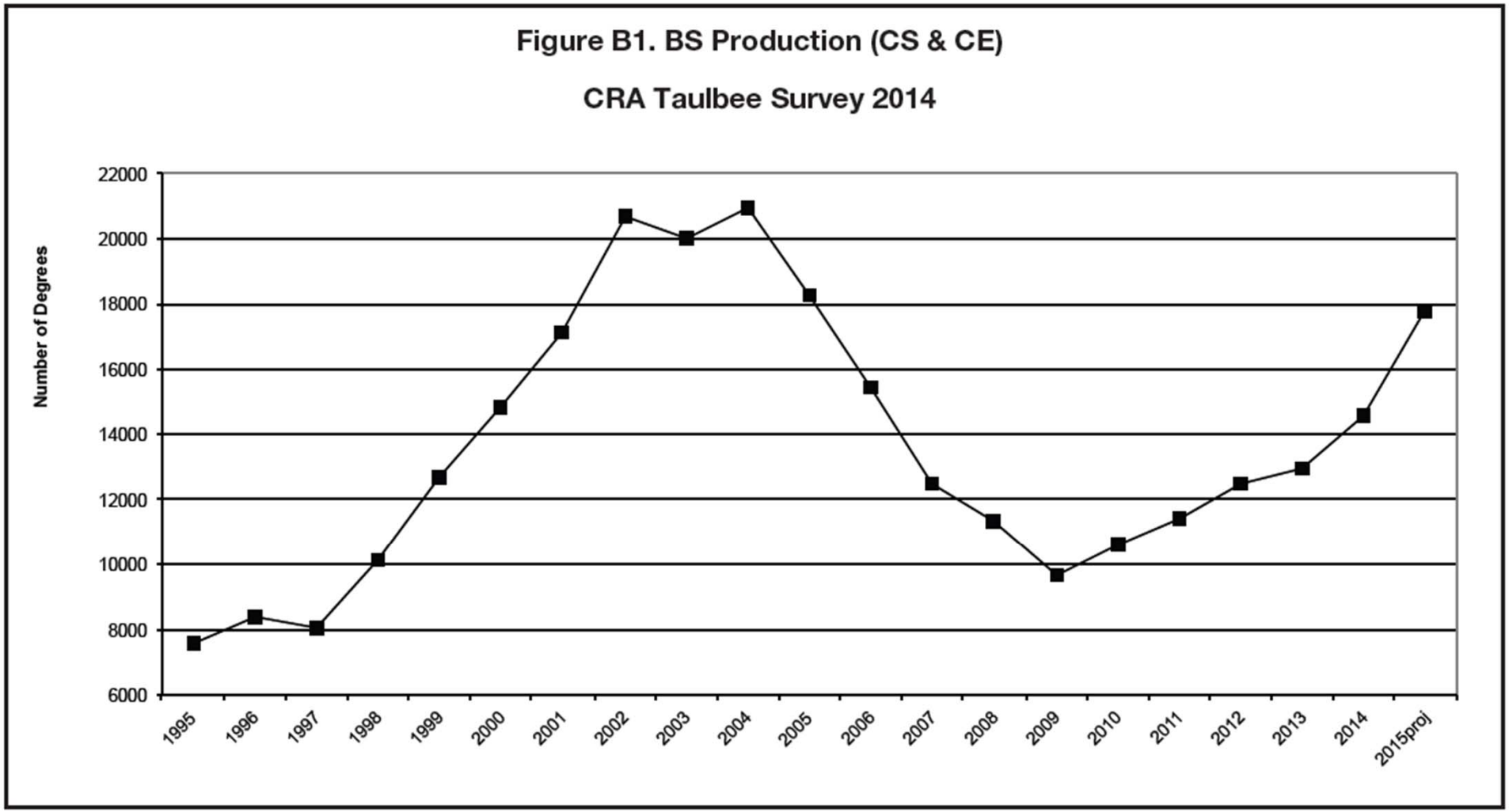


Sources::
Android; Google; App Annie; Business Insider

Additional Information:
Worldwide; December 2009 to February 2015

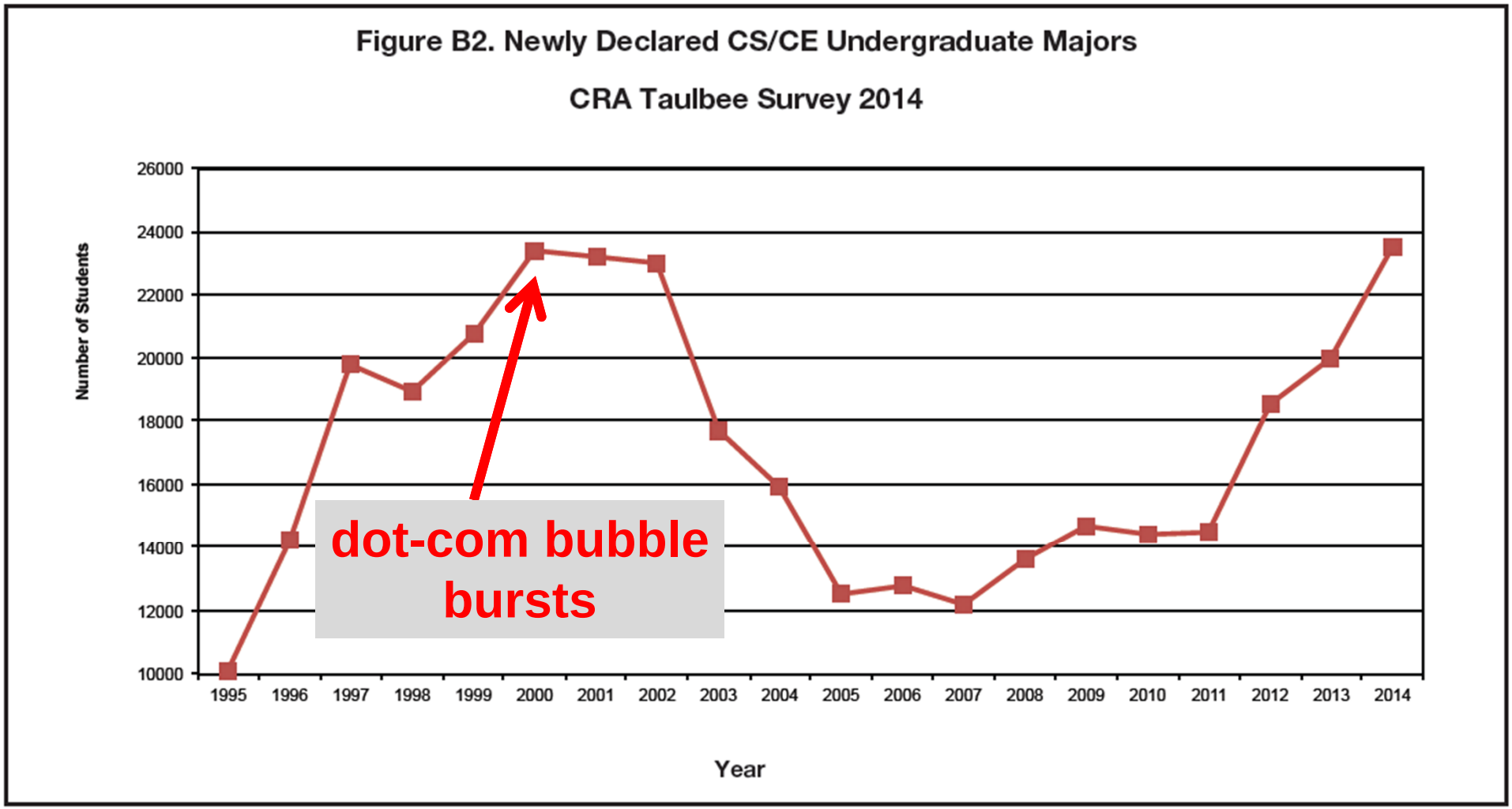
statista

And where are all these programmers?



Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

And where are all these programmers?



Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

What are our defenses?

Education

Secure design, secure coding, and vulnerability assessment need to be a part of our undergrad and grad curriculums

Training

Organizations can't be complacent and leave it to the universities or the employees

Policies

Need requirements and procedures, standards, best practices.

Tools

Static analysis tools offer a first line of defense

Review and Assessment

Red/blue teams are essential to security critical code.

Education

Where is secure coding in our university curriculums?

While most security courses include a week or so on the topic, this is insufficient and not seen by enough students.

Full courses on secure coding, secure design, and software assessment are rare (and often quite incomplete).

Also needs to be integrated with other subjects: e.g., teaching about SQL injections in the undergrad DB course.

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

Professional Training

Where the university curriculum leaves off, the training companies/organizations step in.

Lots of them:

CMU SEI, SANS Institute, Secure Coding Academy, Infosec Institute, AppSec Labs, Denim Group, John Bryce Co.

OWASP, SafeCode.org, David Wheeler, OpenSecurityTraining.info

… and many others.

The commercial courses vary from fair to very good.

Typical one-week course varies from $2000-$4000/student. Often need several courses.

An increasing number of online and free resources in this area.

Training and Education

Takeaway:

So much software today touches the Internet that there are few that have no attack surface. And even the air gap doesn't seem as robust as it did just a few years ago.

Every CS/CE student needs to get a thorough exposure of secure programming in their undergrad studies.

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

A Quick Interlude about Terminology

Language shapes thought, so it helps to be precise:

Weakness: A flaw in your code. A bug.

> There is a standardized taxonomy of weaknesses called the Common Weakness Enumeration (CWE). Very useful.

Vulnerability: A weakness that can be used to allow someone else (an attacker) to make your software do something harmful, such a release information, change information, or gain control of a system.

Exploit: The code or input sequence that made use of the vulnerability to accomplish the harmful goal.

Tools

A mind-numbing proliferation of assessment tools…

Commercial and open source

Commercial tools often easier to use and more polished. Open source tools often frowned upon, though tools like FindBugs (for Java) can be the most popular and useful in their class.

A hugely active business space, with new companies popping up on a regular basis.

Most tools run locally, but several ‘tools as a service” (TaaS) vendors.

One commercial and free TaaS requires a compressed 300MB download of client code, a long extraction time, and an installed size of 750MB.

On a 20 line C program, the client produced a 3 MB file that is then uploaded to the server for reporting/viewing.

Tools

Static code analysis tools:

- Tools for every language and framework.
- Run on source, byte code (including DEX) and binary.
- Static analysis tools are the simplest (not simple) to use because they don't require execution.
- Many tools require your code to build first.
- Often challenging to install (huge: GBs) and run.
- Have to get options set right or you get too many/few reports.
- Each tool reports the results in a somewhat different form. Many translate to CWEs, which is better for comparison.
- Some tools are *fast*, some are **deep**, some are tunable (if you can figure out how), and a few are sound (but slow and specialized).

Different tools are good at different things, so running a single tool may be quite limiting (though the commercial vendors really don't this idea).

Tools

Dynamic code analysis:

Some run on binaries and some require controlled compilation.
Requires an execution environment and representative input data.

- Memory access checks, taint analysis …

Dynamic testing tools

Scan web services for properly configured web servers and well behaving web applications.

Some tools are generic and some need to understand the application.

Need a complete client and server environment running.

¬ Tools

In the real world, there is a lot of resistance to even running the most basic of static analysis tools:

- Commercial tools are expensive.
- Cumbersome to install. Then you have to keep them up to date.
- Reports not always easy to understand for a non-security expert.
- Getting the options set correctly can be tricky.
- One a legacy code base, the false-positives when first running the tool can be exhausting.
- Security flaws slow down software delivery so often better left undiscovered.

Training and Education

Takeaway:

Let's make tools easier to run for users, providing help for both the users and tool builders.

The Software Assurance Marketplace (SWAMP)

An effort funded by DHS S&T for five years …

… at the halfway point of this project

An open and operational facility to advance software assurance.

A consortium of four academic institutions: Morgridge Institute of Research, University of Wisconsin-Madison, Indiana University and the University of Illinois-Urbana.

The Software Assurance Marketplace (SWAMP)

Building a software assurance community …

… a meeting place for technology practitioners and technology users

Raising the bar on the state of software assurance …

- For software developers, lowering the barriers to adopting comprehensive assurance tools.
- For tool developers, simplifying their tasks, enabling new players to compete, providing feedback and guidance
- For researchers, a test bed and source of data
- For instructors, a platform to introduce software assurance.

What Makes the SWAMP "the SWAMP"?

An environment where new software packages can be added easily and efficiently

An environment where new tools can be added easily and efficiently

Support for tools that integrate and interpret the output of SwA assurance tools

An open framework with controlled access to software products and results at all levels

A foundation for understanding the process of software assessment

Help for Both the Novice and Expert

The *newbie* can start using assurance tools with little effort or preparation.

The expert does familiar tasks, but with less effort and more precision.

A comprehensive viewer brings multiple results into a single cohesive view

The Software Assurance Marketplace (SWAMP)

Making assurance *integral* to the software lifecycle:

Continuous Assurance

At each step in the software development, evaluate the software for flaws.

- Like a trajectory, earlier corrections are much more effective and less expensive than the conventional develop-then-test practice.

Making the cultural change:

- Delays in adopting such practices raise the cost, often to prohibitive levels.

The Software Assurance Marketplace (SWAMP)

A suite of 16 software assurance tools for:

- Languages and environments: C, C++, Java, Python, Ruby, Android
- Commercial (Parasoft Ctest and Jtest) and open source tools
- Upcoming release: Red Lizard Goanna (C/C++), GrammaTech CodeSonar (C/C++), Brakeman (Ruby), Dawnscanner (Ruby)
- Soon: Facebook Infer (Java), JSLint (Javascript),JSHint (Javascript), Code Sniffer (PHP), RIPS (PHP), CSS Lint (CSS)

Integrated results viewing

- A single user experience independent of the tool, language, or environment

The automation necessary to streamline the user experience

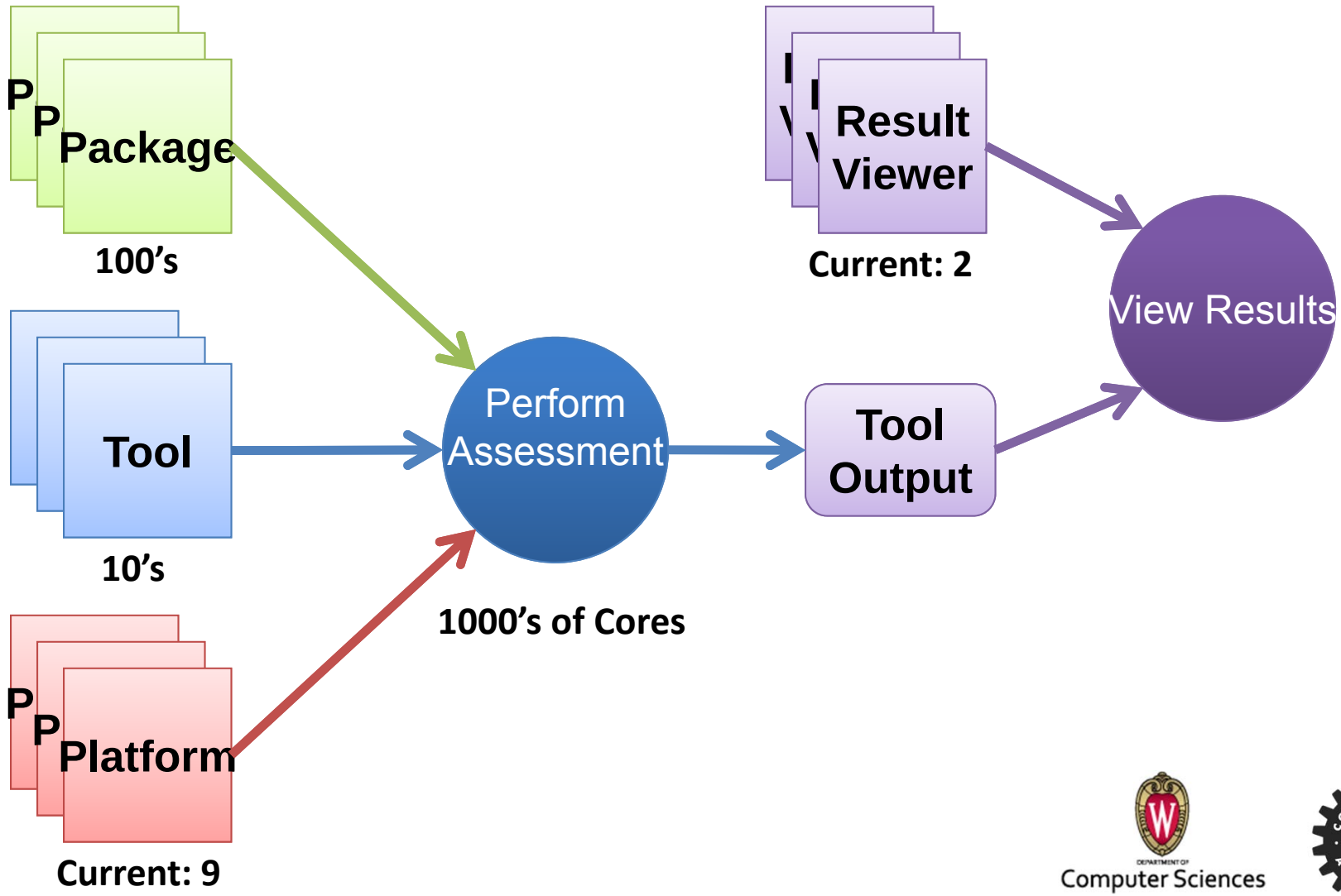
- No fussing with tool settings and options
- Automated integration into your software build
- Direct pulls from code repository
- Direct exports from your IDE

Upcoming:

- Tools for Windows/C#/.NET, MacOS

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

The Software Assurance Market Place



Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

Automation Is Key

The SWAMP offers:

- The automation to run tools easily
  - Applying a tool to a new software package takes little effort
  - Users receive feedback after *each* code update or commit.
- The smarts to automatically combine results in a unified report (thank you SecureDecisions CodeDX)
- The ability to track progress and trends over time.
- A resource for organizations to institute a continuous assurance methodology to meet the needs of a continuous integration world.

Simplistic Static Source Code Assessment

One source file

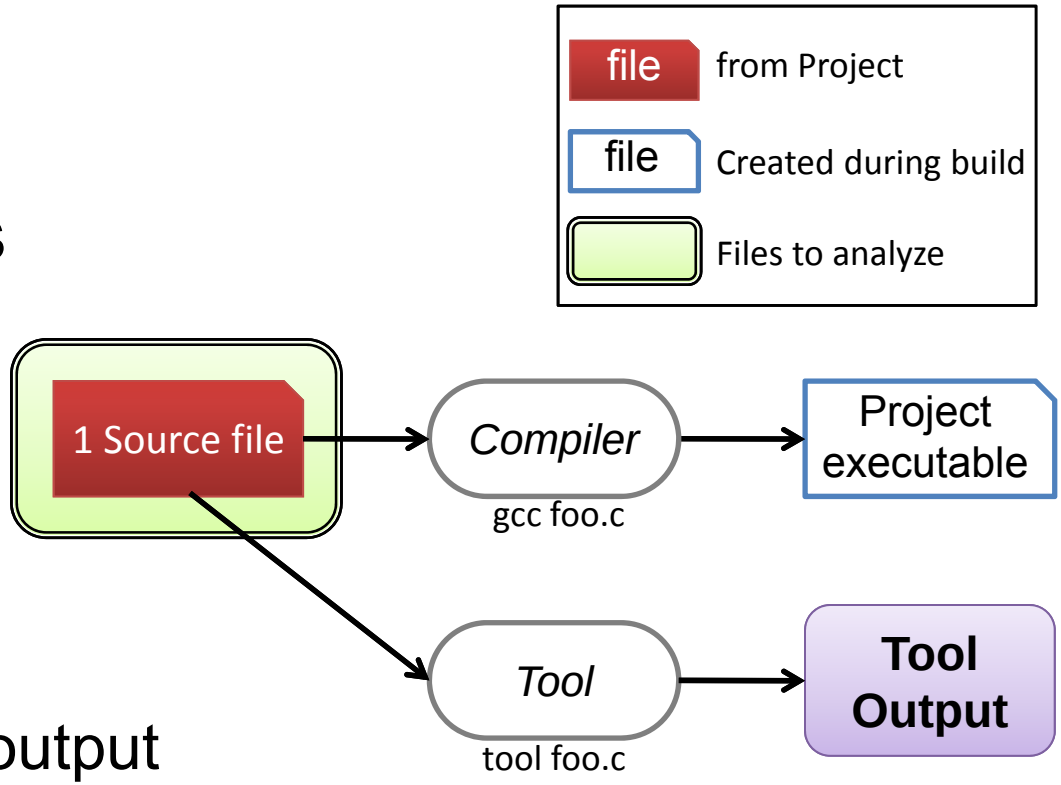
No compiler options

Default libraries

One executable

One tool

- Source as input
- Weaknesses as output
- Run similar to a compiler



Real software is a bit more complex…

Real Software Is More Complex

Multiple source files

- Separate compilation to form object files
- Object files combined to form libraries and executables

Multiple executables

Build generators, multi-level makefiles, custom scripts, complex conditional compiling criteria.

Not obvious what code to assess (no easily obtained list)

- Generated files
- System and 3rd party files outside of project directories
- Command line arguments affect operation of build tools

No standard technology to build (usually contains custom code)

Build contains the information needed to assess, but…

Usually have only minimal understanding of build system

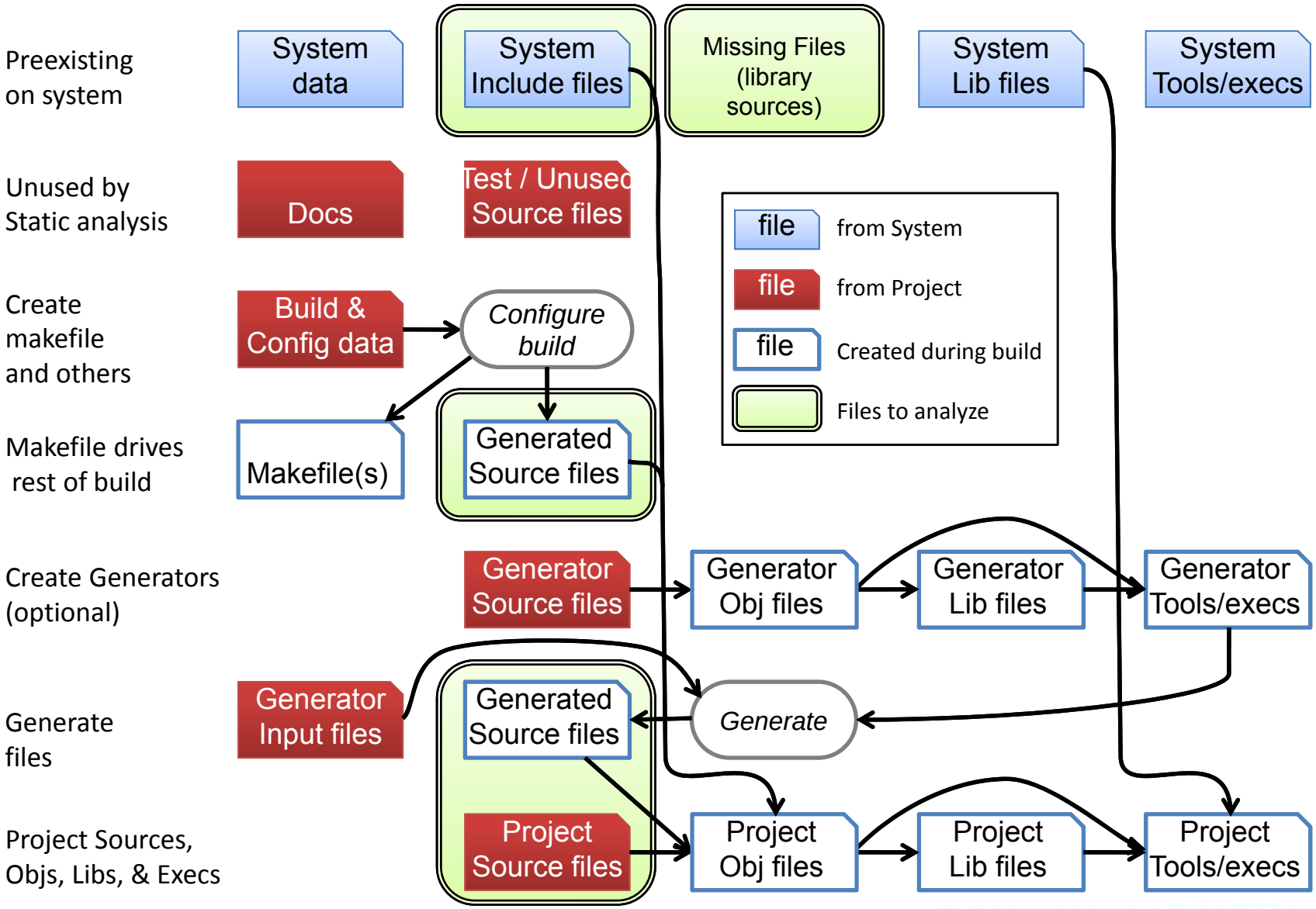
Build description usually

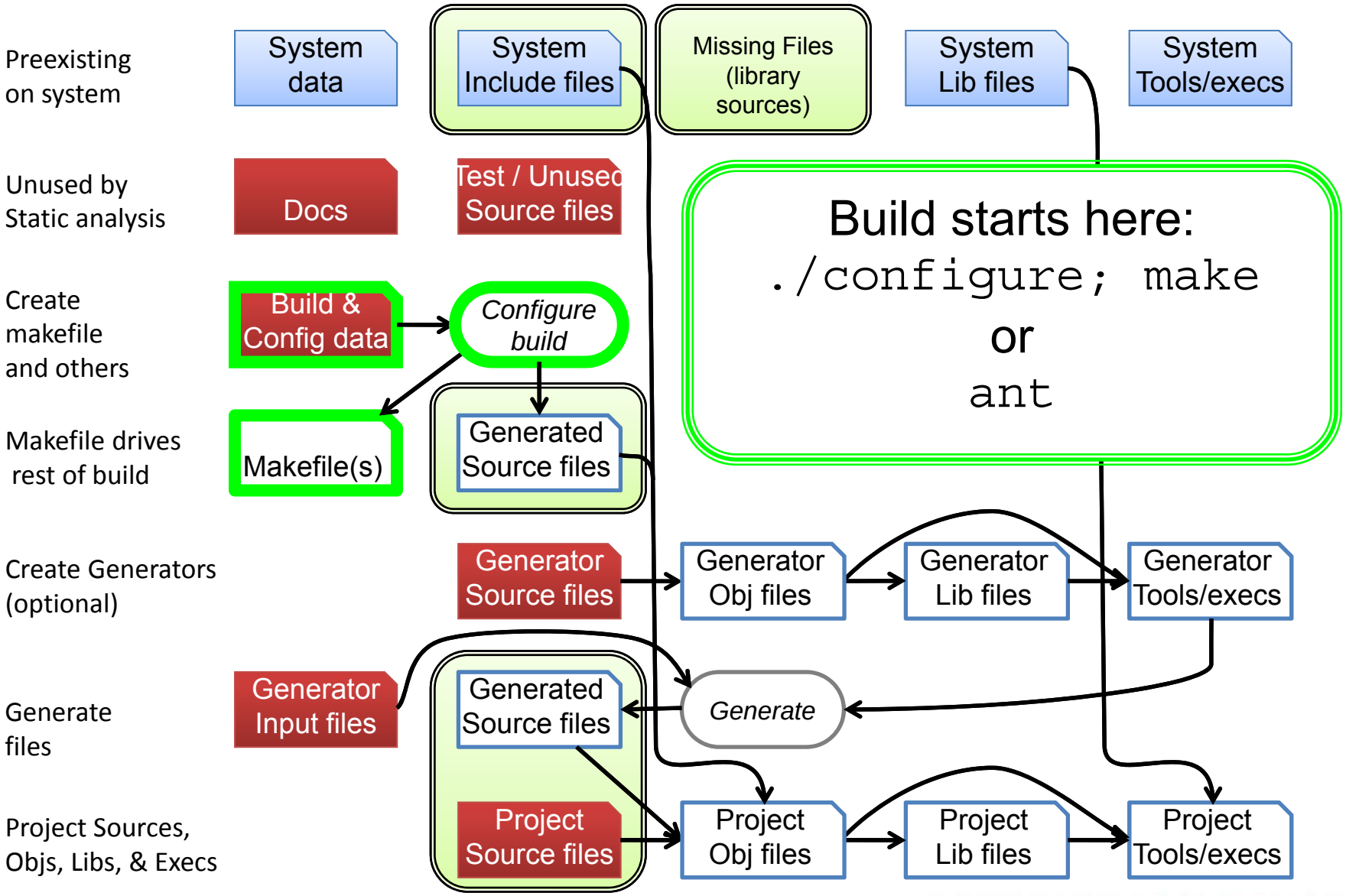
- Partially declarative
- Partially code interpreted at build time

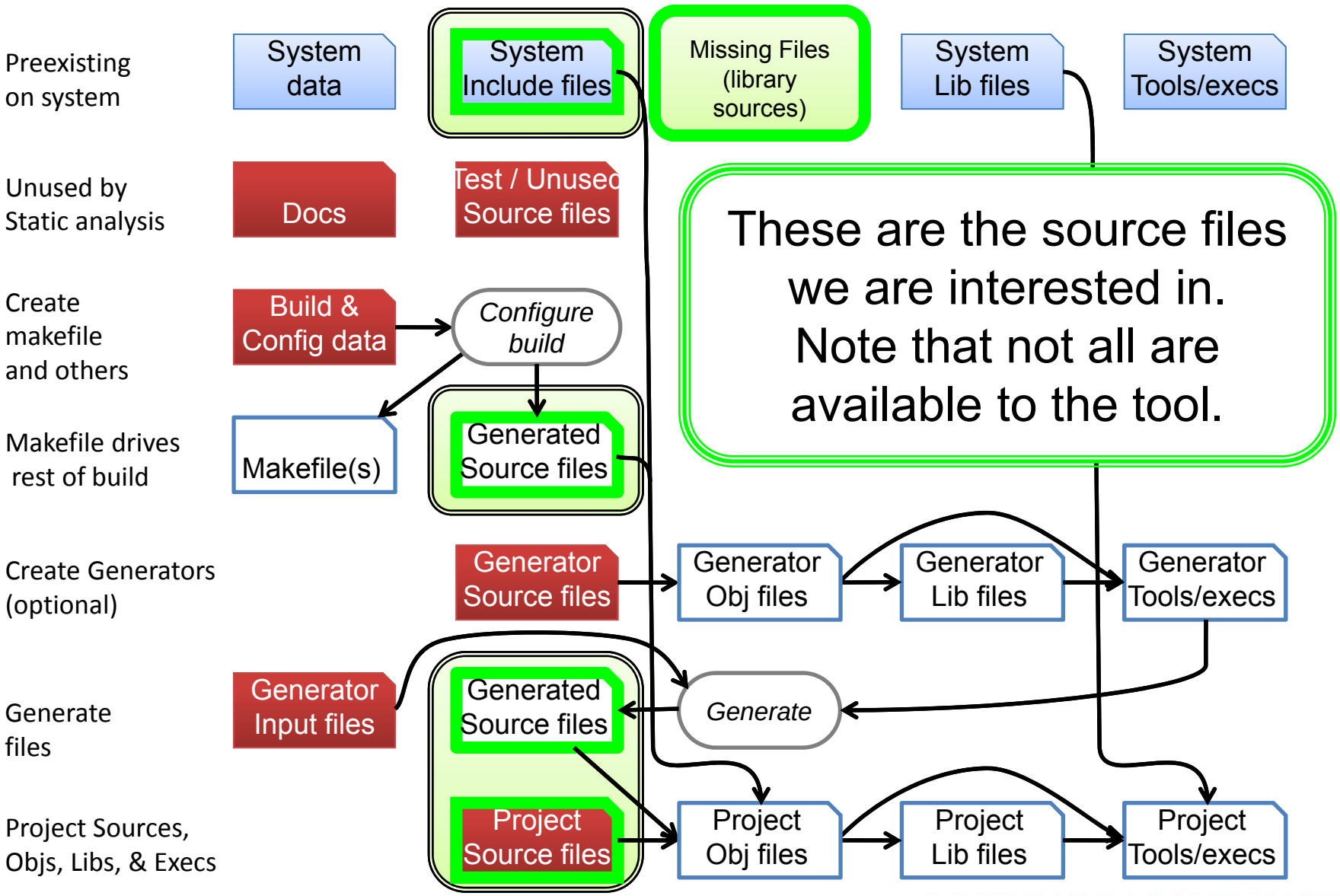
Many arbitrary programs run to build the software with many layers

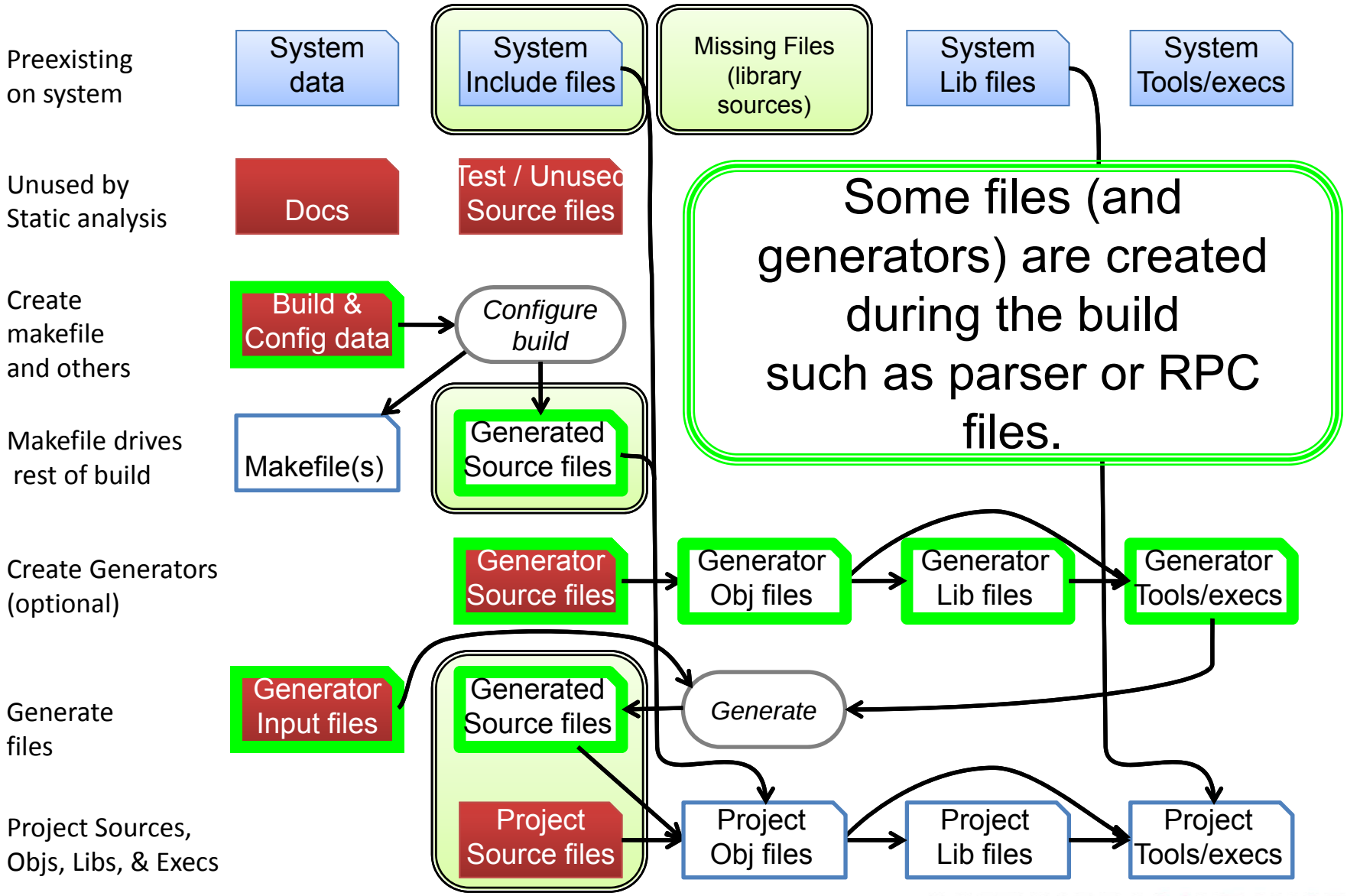
1. Build configuration layer create makefiles
2. Make drives build process
3. Calls shell code snippets
4. Compilation, library creation, linkage of executables

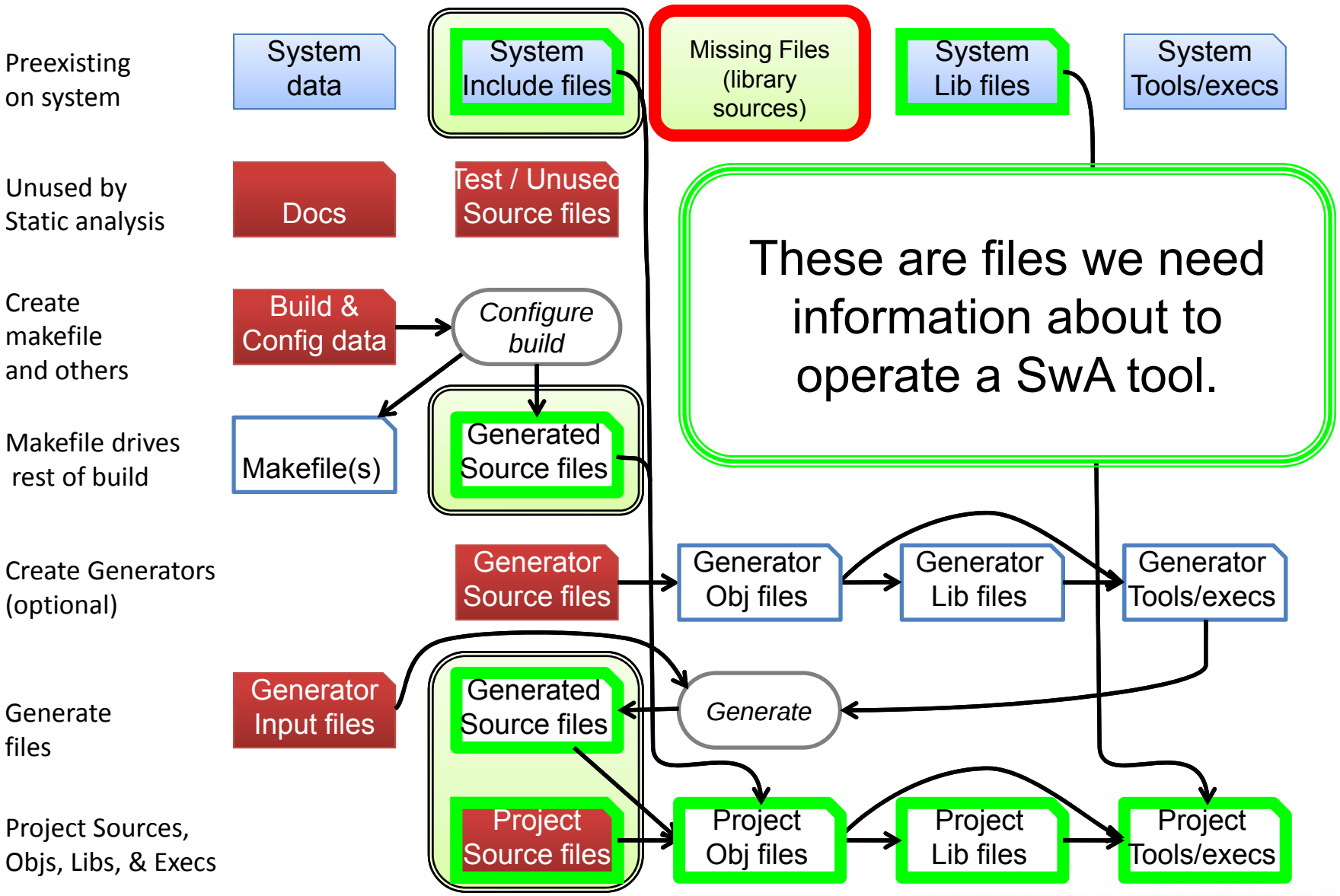
Capturing build tool invocations is non-trivial


Preexisting on system
Unused by Static analysis
Create makefile and others
Makefile drives rest of build
Create Generators (optional)
Generate files
Project Sources, Objs, Libs, & Execs
System data
System Include files
Missing Files (library sources)
System Lib files
System Tools/execs
Docs
Test / Unused Source files
Build & Config data
Configure build
Makefile(s)
Generated Source files
file from System
file from Project
file Created during build
Files to analyze
Generator Source files
Generator Obj files
Generator Lib files
Generator Tools/execs
Generator Input files
Generated Source files
Generate
Project Source files
Project Obj files
Project Lib files
Project Tools/execs


Preexisting on system
Unused by Static analysis
Create makefile and others
Makefile drives rest of build
Create Generators (optional)
Generate files
Project Sources, Objs, Libs, & Execs
System data
System Include files
Missing Files (library sources)
System Lib files
System Tools/execs
Docs
Test / Unused Source files
Build & Config data
Configure build
Makefile(s)
Generated Source files
Build starts here:
./configure; make
or
ant
Generator Source files
Generator Obj files
Generator Lib files
Generator Tools/execs
Generator Input files
Generated Source files
Generate
Project Source files
Project Obj files
Project Lib files
Project Tools/execs


Preexisting on system
Unused by Static analysis
Create makefile and others
Makefile drives rest of build
Create Generators (optional)
Generate files
Project Sources, Objs, Libs, & Execs
System data
System Include files
Missing Files (library sources)
System Lib files
System Tools/execs
Docs
Test / Unused Source files
These are the source files we are interested in. Note that not all are available to the tool.
Build & Config data
Configure build
Makefile(s)
Generated Source files
Generator Source files
Generator Obj files
Generator Lib files
Generator Tools/execs
Generator Input files
Generated Source files
Generate
Project Source files
Project Obj files
Project Lib files
Project Tools/execs


Preexisting on system
Unused by Static analysis
Create makefile and others
Makefile drives rest of build
Create Generators (optional)
Generate files
Project Sources, Objs, Libs, & Execs
System data
System Include files
Missing Files (library sources)
System Lib files
System Tools/execs
Docs
Test / Unused Source files
Some files (and generators) are created during the build such as parser or RPC files.
Build & Config data
Configure build
Makefile(s)
Generated Source files
Generator Source files
Generator Obj files
Generator Lib files
Generator Tools/execs
Generator Input files
Generated Source files
Generate
Project Source files
Project Obj files
Project Lib files
Project Tools/execs


Preexisting on system
Unused by Static analysis
Create makefile and others
Makefile drives rest of build
Create Generators (optional)
Generate files
Project Sources, Objs, Libs, & Execs
System data
System Include files
Missing Files (library sources)
System Lib files
System Tools/execs
Docs
Test / Unused Source files
Build & Config data
Configure build
Makefile(s)
Generated Source files
These are files we need information about to operate a SwA tool.
Generator Source files
Generator Obj files
Generator Lib files
Generator Tools/execs
Generator Input files
Generated Source files
Generate
Project Source files
Project Obj files
Project Lib files
Project Tools/execs

Multiple Executable Difficulties

Usually contains many executables

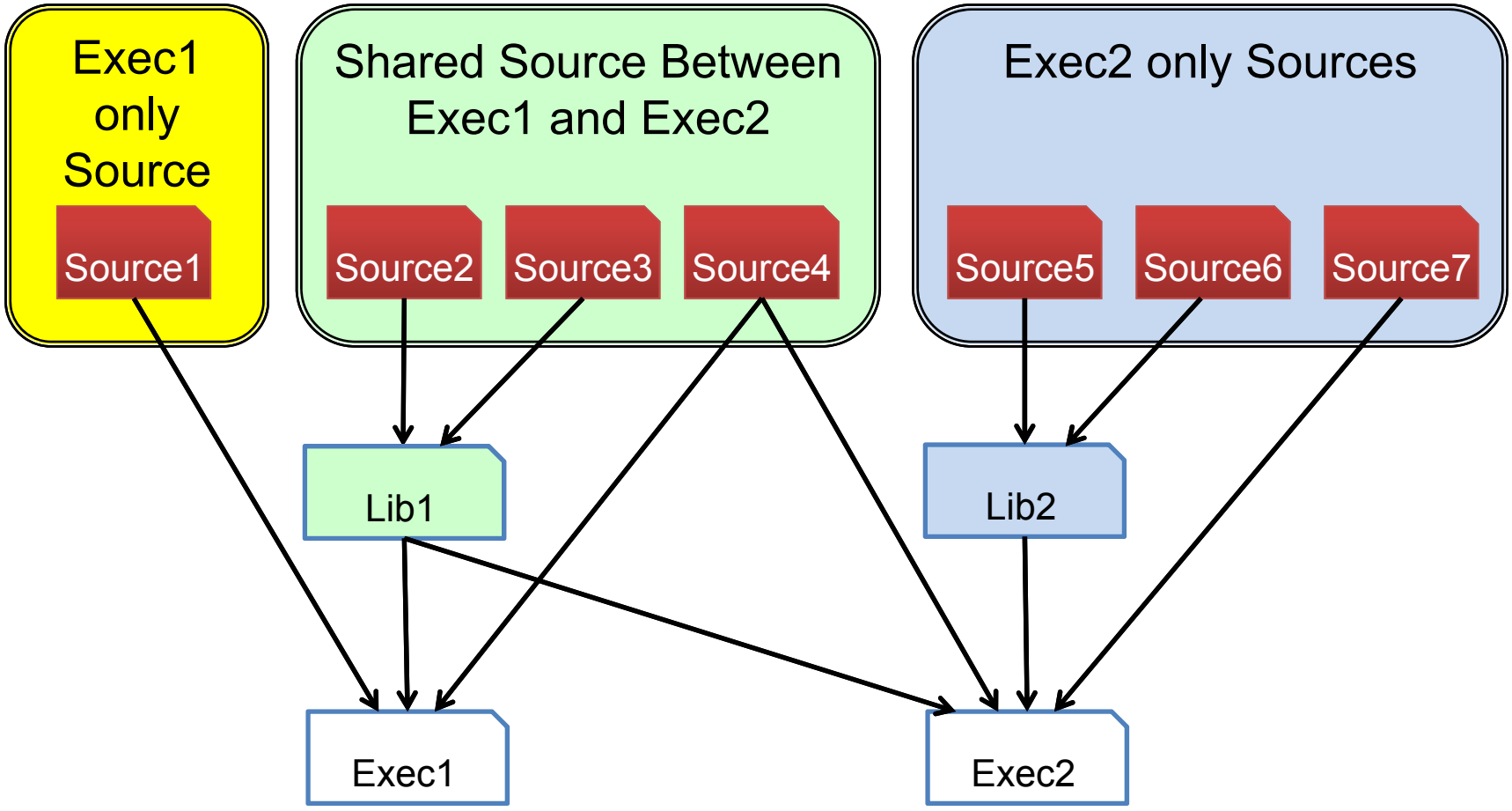
Need to determine:

- What executables exists
- What source files used in build
- Whole package analysis not acceptable
  Must be whole program by executable

Too complex to leave to humans,
requires automated tool

Some tools seem to do whole package
(all the sources compiled in the package)

Multiple Executable per Package



Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

Still More Complexity for C/C++

Need to know command line options

- Macro defines and undefines
- Included file location
- Language dialect options

Need to know current working directory

- Source and library file
- include file location

Need to know environment as it can affect compiler options

Open Source Tools' Approaches for Collect File Information

Process all sources under a directory

- May not be right set of files

Provide instructions/recipe on how to call the tool or modify the build system

- Difficult for developer to modify build correctly

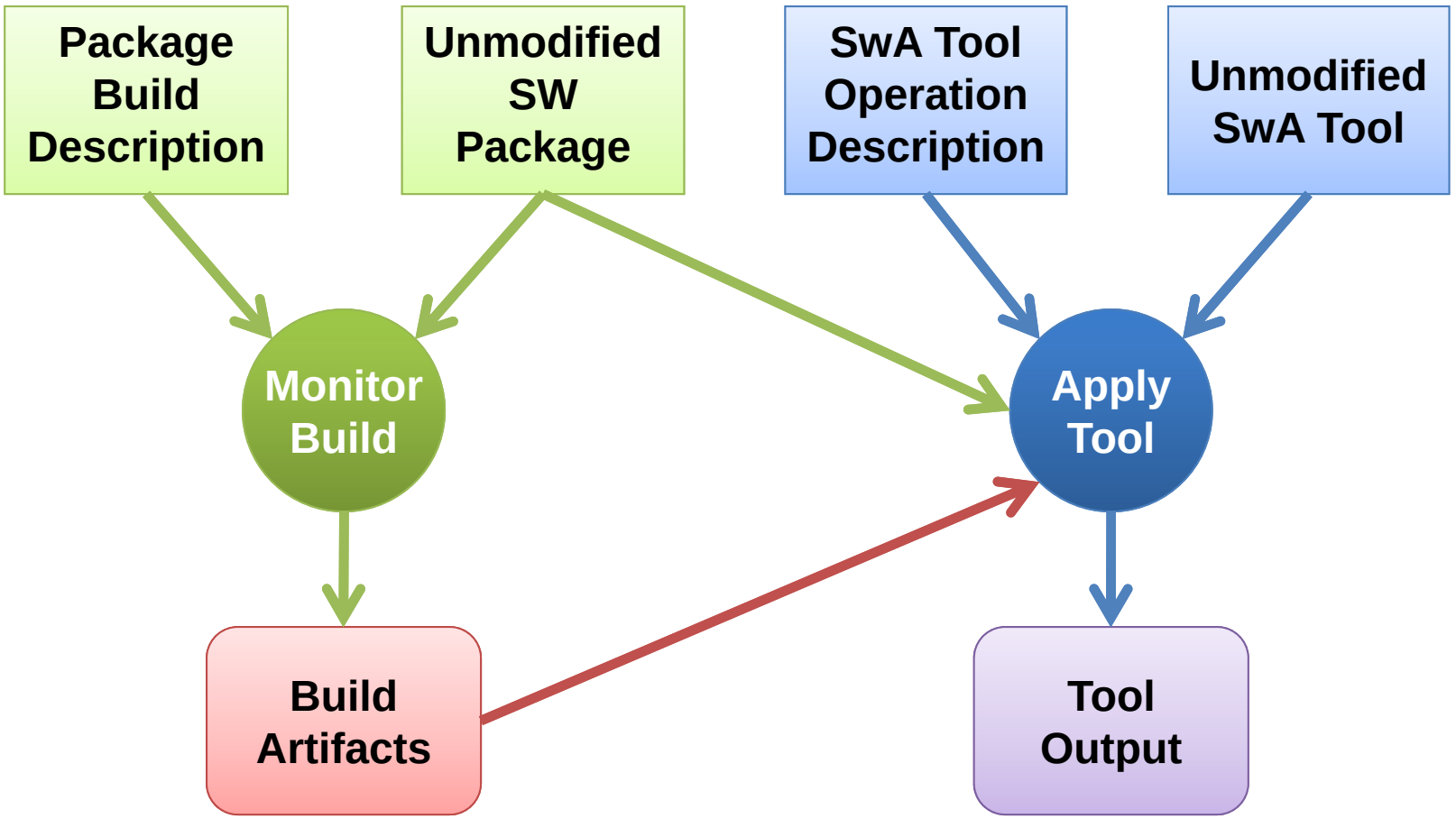
Make assumptions about build environment to insert a shim to capture relevant build operations

- Fragile if assumptions are violated

Ignore aspects of build such as command options and environment

- Tool operates on different code than compiled

An Automated Approach to Assessments:
Monitor the build, apply the tool



Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

Current State of the SWAMP

Have generic tool to determine build artifacts

- Any build process for C/C++
- Ant, Maven, Gradle builds for Java and Android

Captures executable invocation of build process

- Command path
- Full arguments
- environment
- Current directory path

Applies tool using build artifacts and SwA Tool Operation Description

Open source: free to you

Key Attributes

Highly automated

- If you can compile your tool in the SWAMP, all else is automated.

Secure

- Strong sandboxing: all executions in single-use virtual machines

Private (if you wish)

- Share your tool, app, or data if and when you choose.

Open

- Lots of tools, lots of apps, lots of assessment data

A resource

- Software to help make your job easier; people to advise you

A community

- Join and leverage other like-minded users online and in person.

The Software Assurance Marketplace (SWAMP)

Beyond a single facility …

… a model for organizations to establish their own facilities.

It's all open source …

. . . and, while open source isn't free (someone has to develop it), it is freely shared and used.

A constellation of SWAMPs …

- Private SWAMPs for companies that want to control access.
- Foreign partner SWAMPs when software can't cross national boundaries.
- "Highside" (classified) SWAMPs

As a resource to evaluate and prototype new regulatory strategies.

As a path to software certification standards.

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

CONTINUOUS ASSURANCE

It's not all wine and roses though …

Getting an organization to use a tool, any tool, let alone a comprehensive service is a battle.

"What?"

"No thanks, we don't really need it."

"Oh, right. But we really can't afford it."

"Didn't we take care of that already?"

Tools need some serious improvements (that's where many of you come in…)

They're either too slow, too limited, or too hard to understand.

We don't really know how good these tools are.

And, even if we know, we can't talk about it.

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

It's not all wine and roses though …

Getting them to use a tool:

Software assurance often falls into the testing group. And these folks typically don't understand security concerns, so don't think that it's important. Even in 2015. (Really.)

Even if you convince them they need it, they either can't afford the effort to install and maintain tools, or the time to train programmers in their use, or the delay such testing will add to the release cycle.

And if they think such tools make sense, they might think that they can't afford them.

Besides, didn't we just do a risk assessment, so we're good, right?

It’s not all wine and roses though …

And the tools themselves are far from perfect:

How do we know how good are the tools? No one publishes comparison studies any more.

There are calibration test suites produced by NSA/NIST and IARPA.

It's not all wine and roses though …

Juliet:

- 65,000 small, artificially constructed, micro tests combined into a large amalgamated test program covering 230 CWEs (C/C++ and Java) and varying levels of code complexity.
- 286 directories, each containing between 3-100 micro tests.
- Each test has weakness labeled as a true positive and false positive.
- Produced by NSA and now maintained by NIST

STONESOUP

- 16 large, real world Java and C/C++ programs (e.g., OpenSSL, PostgreSQL, Wireshark; Apache Lenya, Jmeter, Google Web Toolkit).
- Each program is mutated to contain a subset of 78 CWEs.
- 8500+ complete programs to test.
- Directory of labelings of functions that contain weakness.
- Compare tool results from unmodified program to the mutated one.
- No general facility for testing false positives.

Now, it we were only be able to publish these results!

Legal Hazards to Our Security

In 2009, we did an interesting study.

- We started with 15 serious exploitable (and exploited) weaknesses in a large, complex production code base (the HTCondor system). These came from our blue team in-depth assessment activities.
- We carefully identified and labeled the source code location and cause of each weakness.
- We ran what were considered to be the best-of-breed tools of the day: Fortify SCA 5.1 and Coverity Prevent 4.1.
- Fortify found 6 and Coverity found 1 of the 15 serious weaknesses.
- We published it, along with analysis of false positives (lots!)

  James A. Kupsch and Barton P. Miller, "Manual vs. Automated Vulnerability Assessment: A Case Study", *First International Workshop on Managing Insider Security Threats (MIST 2009),* West Lafayette, IN, June 2009.
- Five years later, Coverity went legal on us. To no avail.

We were lucky then; you can't do this anymore … 

Legal Hazards to Our Security

The shrink-wrapped license, the EULA, containing the *Dewitt Clause:* you can only publish what we allow they about their software. Sigh.

This first came to notoriety when UW Prof. David Dewitt published a benchmarking study of several database systems, including Oracle, in the 1983 VLDB.

Larry Ellison called our CS chair and demanded that Dewitt be fired. Following that event, Oracle (and now, many others) added a clause to their user agreement forbidding such reports.

Legal Hazards to Our Security

Example 1:

2.0 GRANT OF LICENSE AND USE OF SOFTWARE

...

2.2 **Usage Rights**. ... You shall not ... (e) perform, publish, or release to any third parties any benchmarks or other comparisons regarding the Software or User Documentation.

Legal Hazards to Our Security

Example 2:

2 Limitations on Software Use. You may not:

…

2.2 disclose Software output, including by not limited to the results of any benchmark test of the Software, or Software documentation to any third party without XXX's prior written approval;

Legal Hazards to Our Security

Example 3:

2. 2 ***Conditions.*** The rights granted to Customer above are conditional upon Customer's compliance with the following obligations:

...

Customer will not disclose to any third party any comparison of the results of operation of XXX' Licensed Products with other products.

Legal Hazards to Our Security

Not to mention:

- Limited liability clauses
- No warrantee clauses
- No reverse engineering clause (thanks Loren)

We had a chance to present this to the Senate Homeland Security and Government Affairs Committee.

And to some public minded intellectual property rights lawyers.

Stay tuned….

Legal Hazards to our Security

Takeaway:

The lack of transparency in the capabilities of these software tools is hurting our national and international security.

Some Fun/Bad Things that We (and others) Found

In the course of our blue team engagements, we've found some interesting and tricky vulnerabilities that were beyond the scope of tools.

Of course, there have been notable vulnerabilities, such as Heartbleed, found by others.

So, here's a bit of discussion ….

Code Injection Vulnerability

1. logfile – name's value is user controlled

```
name = John Smith
name = ');import os;os.system('evilprog');#
```

Read logfile

2. Perl log processing code – uses Python to do real work

```
%data = ReadLogFile('logfile');
PH = open("|/usr/bin/python");
print PH "import LogIt\n";w
while (($k, $v) = (each %data)) {
  if ($k eq 'name')  {
  print PH "LogIt.Name('$v')";
}
```

Start Python, program sent on stdin

3. Python source executed – 2nd LogIt executes arbitrary code

```
import LogIt;
LogIt.Name('John Smith')
LogIt.Name('');import os;os.system('evilprog');#')
```

Code Injection Vulnerability

Cleverness should definitely be punished.

Dynamic code generation, while elegant, offers too rich of an attack vehicle.

This is one that tools should really be able to find.

Code Injection Mitigated

1. logfile – name's value is user controlled

```
name = John Smith
name = ');import os;os.system('evilprog');#
```

2. Perl log processing code – use QuotePyString to safely create string literal

```
%data = ReadLogFile('logfile');
PH = open("|/usr/bin/python");
print PH "import LogIt\n";w
while (($k, $v) = (each %data)) {
  if ($k eq 'name')  {
    $q = QuotePyString($v);
    print PH "LogIt.Name($q)";
}
```

```
sub QuotePyString  {
  my $s = shift;
  $s =~ s/\\/\\\\/g;     # \  ➔ \\
  $s =~ s/'/\\'/g;       # '  ➔ \'
  $s =~ s/\n/\\n/g;      # NL ➔ \n
  return "'$s'";         # add quotes
}
```

3. Python source executed – 2nd LogIt is now safe

```
import LogIt;
LogIt.Name('John Smith')
LogIt.Name('\');import os;os.system(\'evilprog\');#')
```

Command Argument Injection Example

- **Example**

```
snprintf(userMsg, sSize, "/bin/mail -s hi %s", email);
M = popen(userMsg, "w");
fputs(userMsg, M);
pclose(M);
```

- **If email is `-I` , turns on interactive mode …**
- **… so can run arbitrary code by if userMsg includes: `~!cmd`**

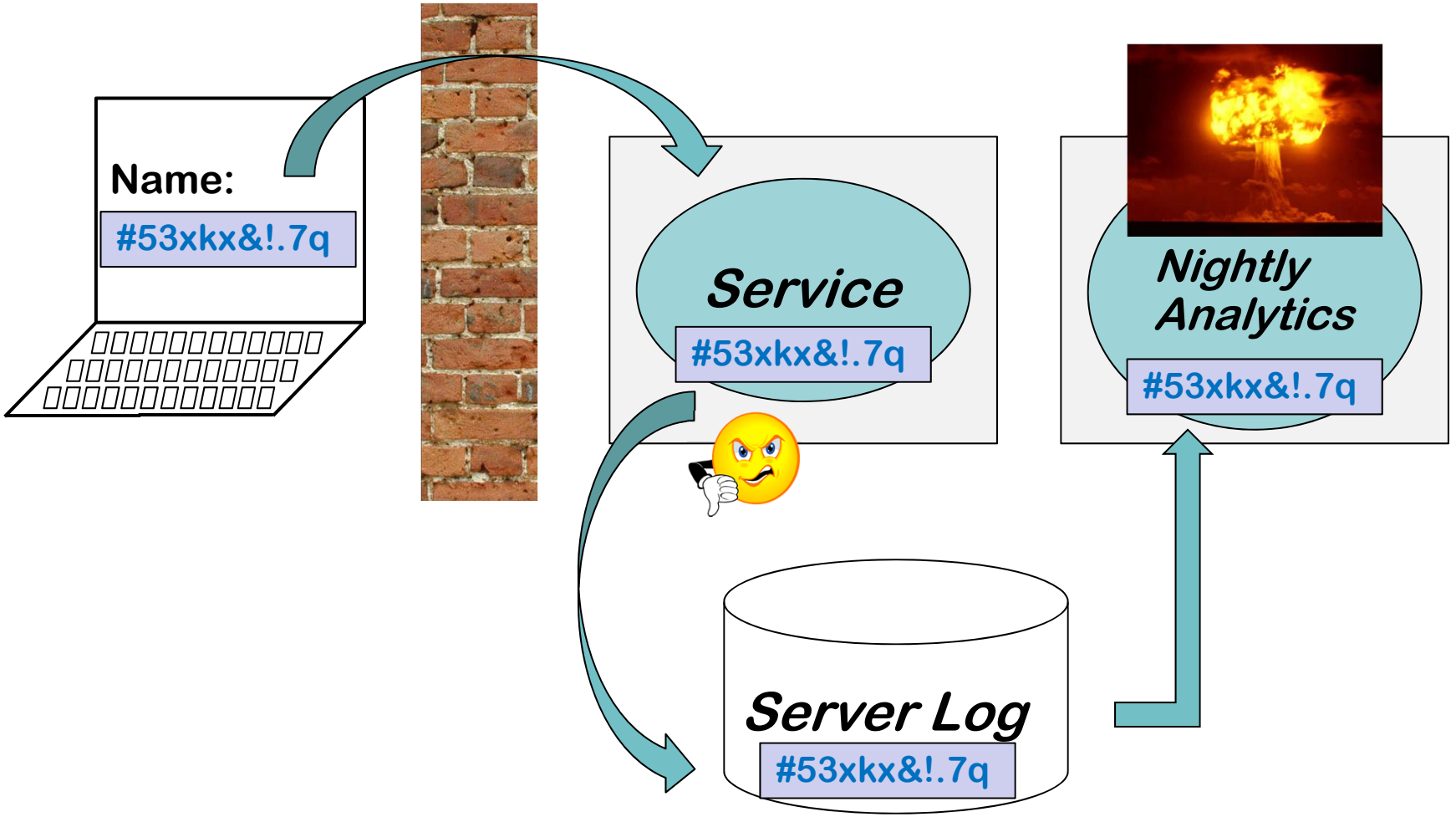
Command Argument Injection Example

Old code rarely goes away. It's hard, expensive, or just too scary to remove legacy code stacks.

Old interfaces were rarely written with the attacker in mind.

It's worked for so long, so well, that we forget that we're even using these interfaces.

Flows Through a File Vulnerability



Flows Through a File Vulnerability

If a single program wasn't hard enough to analyze, tracking flows through series of programs is harder … but critical.

Similar issue comes up in the nuclear simulations. These involve complex flows of data from one program to another. When the input changes (or is found to be erroneous), which results are affected?

Think of it as taint analysis on a grander scale.

Flows and More than One Program

Takeaways:

Our vision has to be at a system scale.

Programs don't live in isolation.

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

Heartbleed

At it's heart (sorry), it's just a buffer overflow…

- Failure of the OpenSSL library to validate the heartbeat packet length field (as compared to the size of the actual message).
- Heartbeat packets are contained within TLS packets.
- The heartbeat protocol is supposed to echo back the data sent in the request where the amount is given by the payload length.
- Since the length field is not checked, `memcpy` can read up to 64KB of memory.

```
memcpy(bp, pl, payload);
```

Destination. Allocated, used, and freed. OK.

Source. Buffer with the heartbeat record. Improperly used.

Length field. Supplied by an untrusted source.

Heartbleed

At it's heart (sorry), it's just a buffer overflow…

- **Failure of the OpenSSL library to validate the length field (as compared to the size of the actual message).**
- **The heartbeat protocol is supposed to echo back the data sent in the request where the amount is given by the payload length.**
- **Since the length field is not checked, `memcpy` can read up to 64KB of memory.**

Heartbleed

Added length check to remediate:

```
if (1+2+payload+16 > s->s3->rrec.length)
        return 0 // silently discard
```

And none of the current tools could fine the problem…why?

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

1. Find the heartbeat packet in the (untrusted) user request

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

2. Extract **user-stated** payload length of the heartbeat packet

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

3. `pl` is an *alias* for the heartbeat packet start address.

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

4. Length of TLS packet that contains heartbeat packet

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

5. payload length **should be** ≤ TLS record length-19

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

6. allocate enough memory for echo packet (according user payload)

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

7. Copy heartbeat data based on the length they claimed. Can also grab other nearby data.

Here's the offending code, slightly redacted

```
2556 unsigned char *p = &s->s3->rrec.data[0], *pl;

2563 n2s(p, payload);
2564 pl = p;
2565
2566 if (s->msg_callback)
2567    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2568               &s->s3->rrec.data[0], s->s3->rrec.length,
2569               s, s->msg_callback_arg);
2570
2571 if (hbtype == TLS1_HB_REQUEST) {
2573    unsigned char *buffer, *bp;
2574    int r;

2580    buffer = OPENSSL_malloc(1+2+payload+padding);
2581    bp = buffer;

2584    *bp++ = TLS1_HB_RESPONSE;
2585    s2n(payload, bp);
2586    memcpy(bp, pl, payload);
```

Need to actually know that `payload` is not trusted (tainted) data.

Heartbleed

Conceptually, this is just an exercise in taint analysis. We need to following the original enclosing TLS packet from a socket, marking it as tainted.

One vendor "fixed" their tool by noting that extracting the integer payload length from a network byte-order uses a byte-swap instruction on a little endian machine, and such a swap instruction is rare enough that this is a sign that the data comes from the network.

Another company could do the taint analysis starting at socket buffers, but didn't do it because it was too slow in practice. When they turned it on *for the right section of code,* it found the problem.

Heartbleed

Takeaways:

Fast, deep, and sound? Is there a path to this goal?

We should stop being surprised by things like Heartbleed. Clearly, languages from the 1970's are killing us. So many lovely new language. What will it take to move systems software away from C/C++?

For the present, how about tools that tell you which code they can't analyze, so you can change your ugly C code to understandable code?

Computer Sciences
UNIVERSITY OF WISCONSIN — MADISON

It's Your Turn

"But if memories were all I sang, I rather drive a truck"

(Ricky Nelson)

If I've left you out, I want to hear from you.

Questions? Thoughts?