

Security Testing

Davide Balzarotti

davide@iseclab.org

Administrative News

Challenge 1 – 36 students (3 in the first 24h)

Challenge 2 – 37 students (6 in the first 24h)

Challenge 3 – 34 students (5 in the first 24h)

Challenge 4 – 17 students (0 in the first 24h)

Challenge 5 (and last) – out today at 3pm

Exam: June 24th

Homework deadline: June 24th

Outline

- So far, we mostly focused on the offensive approach (exploit flaws)
 - Buffer overflow, web vulnerabilities, race conditions
 - Today ... defensive approach – avoid making these mistakes
 - Terminology
 - types of (security relevant) test methods
 - static testing
 - dynamic testing
 - penetration testing
 - Automatic testing: tools, tools, tools...
-

Overview

- When system is designed and implemented
 - correctness has to be *tested*
 - Different types of tests are necessary
 - validation
 - is the system designed correctly?
 - does the design meet the problem requirements?
 - verification
 - is the system implemented correctly?
 - does the implementation meet the design requirements?
 - Different features can be tested
 - functionality, performance, *security*
-

Testing

- Edsger Dijkstra

Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

- Testing
 - analysis that discovers what *is* and compares it to what *should be*
 - should be done throughout the development cycle
 - necessary process
 - but it is not a substitute for sound design and implementation
 - for example, running public attack tools against a server cannot prove that service is implemented securely
-

Testing

- Classification of testing techniques
 - static testing
 - check requirements and design documents
 - perform source code auditing
 - theoretically reason about (program) properties
 - cover a possible infinite amount of input (e.g., use ranges)
 - no actual code is executed
 - dynamic testing
 - feed program with input and observe behavior
 - check a certain number of input and output values
 - code is executed (and must be available)
-

Testing

- Classification of testing techniques
 - white-box testing
 - The tester knows the internals of the system (often the code)
 - testing the entire implementation
 - path coverage considerations
 - find implementation flaws...
...but cannot guarantee that specifications are fulfilled
 - black-box testing
 - The tester does not know the internals of the system and only analyze it from the “outside”
 - testing against specification
 - only concerned with input and output
 - specification flaws are detected...
... but cannot guarantee that implementation is correct
-

Testing

- Automatic testing
 - testing should be done continuously
 - involves a lot of input, output comparisons, and test runs
 - therefore, ideally suitable for automation
 - testing hooks are required, at least at module level
 - nightly builds with tests for complete system are advantageous
 - Regression tests
 - test designed to check that a program has not "regressed", that is, that previous capabilities have not been compromised by introducing new ones
-

Testing

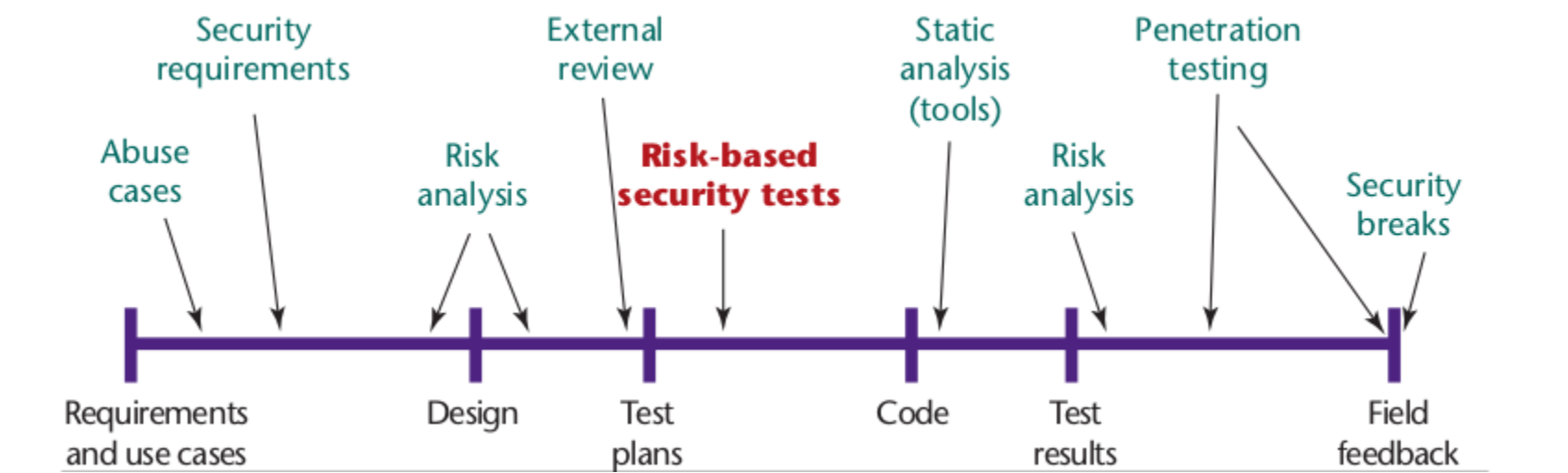
- Software fault injection
 - go after effects of bugs instead of bugs
 - reason is that bugs cannot be completely removed
 - thus, make program fault-tolerant
 - failures are deliberately injected into code
 - effects are observed and program is made more robust
-

Security Testing

- Standard software testing is only concerned with what happens when software fails, regardless of the intent
 - In **security testing**, the difference is the presence of an intelligent adversary bent on breaking the system
 - Security testing involves
 - Testing the security functionalities of the system
(functional security test)
 - Testing the rest of the system to find vulnerabilities that can undermine its security
(vulnerability discovery)
 - Some techniques are predominantly manual, requiring an individual to initiate and conduct the test. Other tests are highly automated
-

Security Testing

- Testing must happen at all different development cycle phases
 - Test method depends on development phase
- Requirements analysis phase
- Design phase
- Implementation phase
- (pre-)Deployment phase



Security Testing – Requirements Analysis Phase

- Software / System requirements usually include only functional requirements
 - security requirements are often omitted
 - If a feature's security requirements is not explicitly stated, ...
 - ... it will not be included / considered in the design
 - ... the programmers will not implement it
 - ... it will not be tested
 - the system will be insecure *by design*
 - Describe how system reacts to exceptional / attack scenarios
-

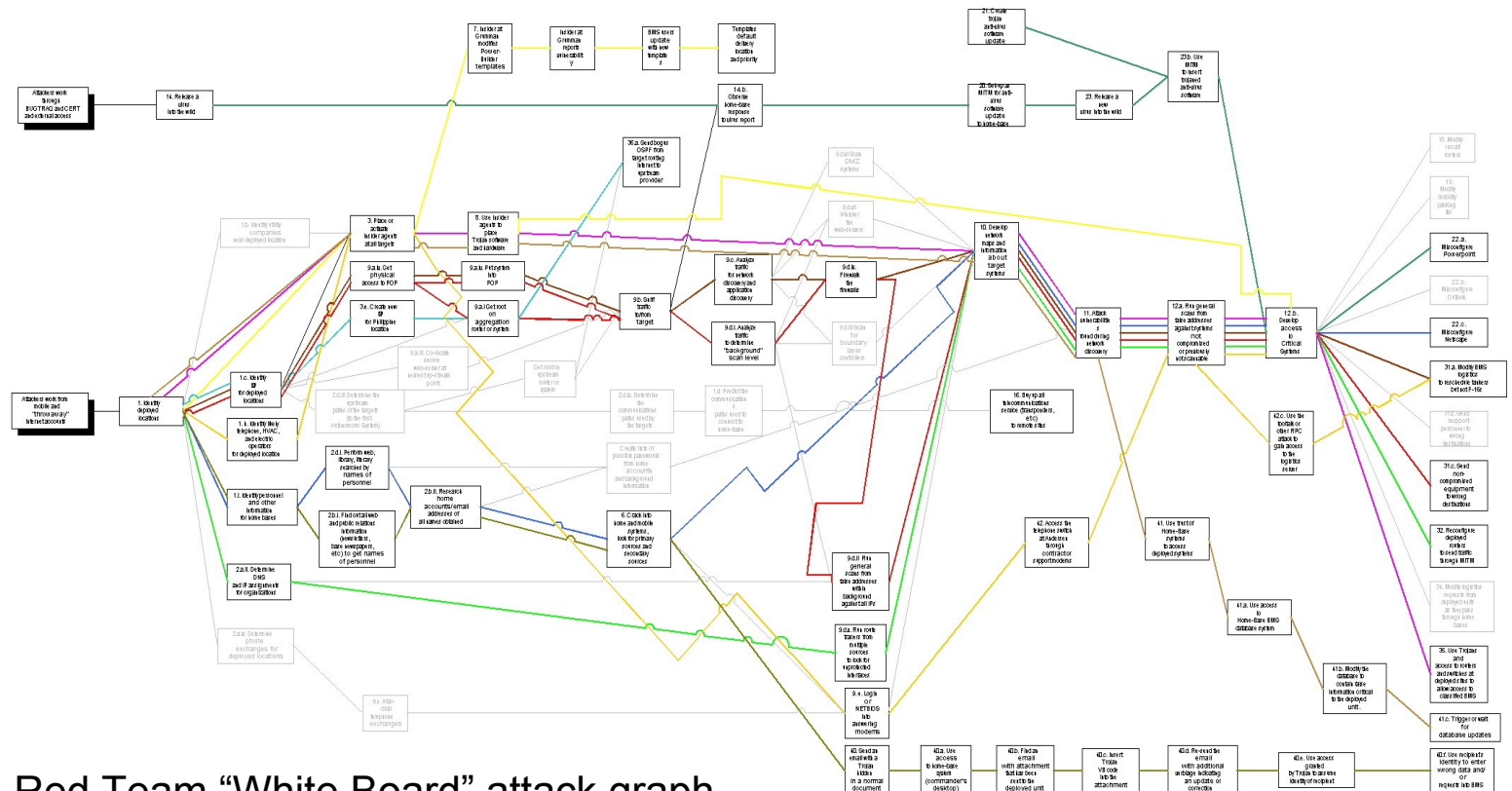
Security Testing – Design Phase

- Design level
 - not much tool support available
 - manual design reviews
 - formal methods
 - attack graphs
 - Formal methods
 - formal specification that can be mathematically described and verified
 - require a model of the systems, of the security properties, and of the attacker capabilities
 - often used for small, *safety*-critical programs / program parts
 - e.g., control program for airplanes
 - e.g., cryptographic protocols
 - state and state transitions must be formalized and unsafe states must be described
 - *model checker* can ensure that no unsafe state is reached
-

Security Testing – Design Phase

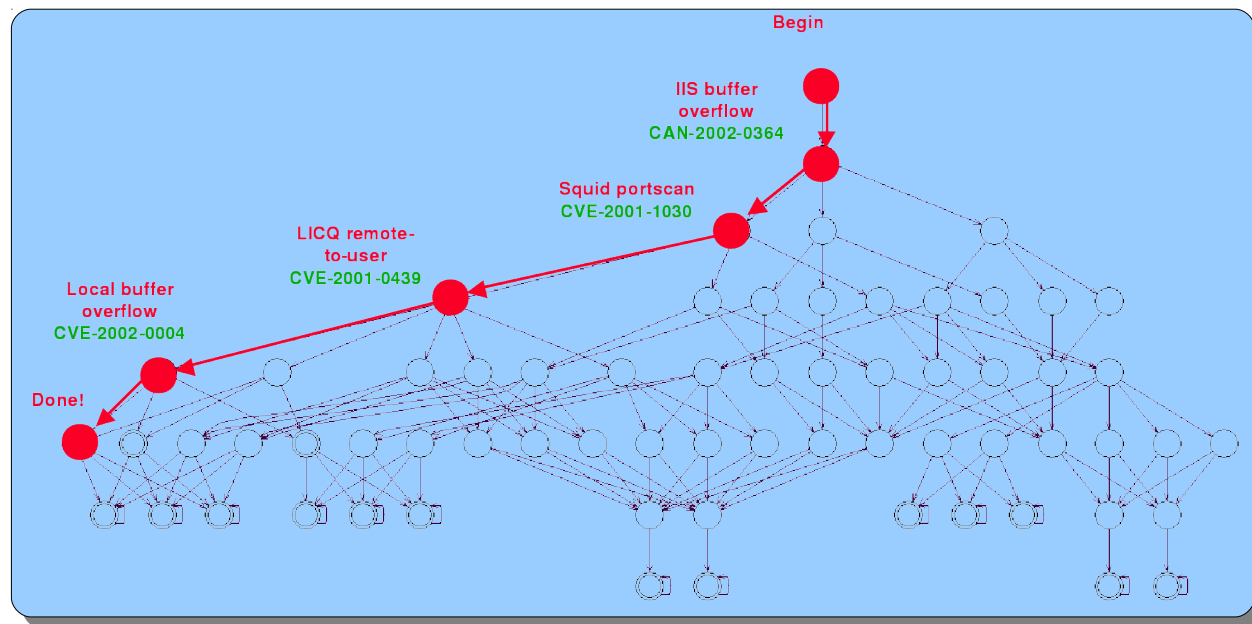
- Attack graphs
 - depict ways in which an attacker can break into a system
 - capture the possible sequences of steps required to reach a certain goal
 - at design phase, can be used to document the security risks of a certain architecture
 - Attack graph generation
 - done by hand
 - error prone and tedious
 - impractical for large systems
 - automatic generation
 - provide state description
 - transition rules
-

Sandia Red Team “White Board” attack graph
from DARPA CC20008 Information battle space
preparation experiment (drawn by hand)



Security Testing – Attack Graphs

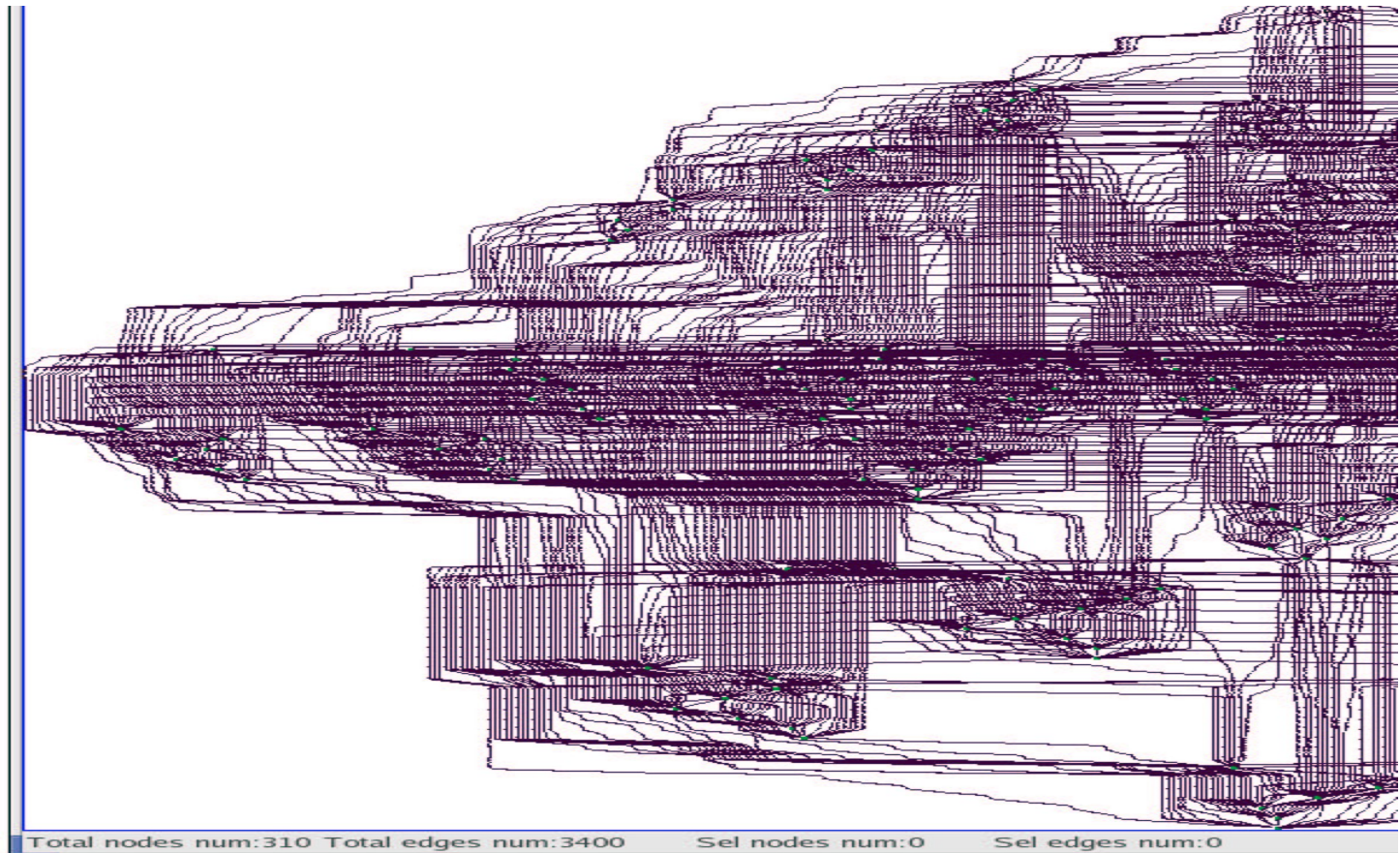
Security property (LTL):
 $G(\text{intruder.privilege}(\text{host}) < \text{root})$



Security Testing – Attack Graphs

P = Attacker gains root access to Host 1.

4 hosts
30 actions
310 nodes
3400 edges



Security Testing – Implementation

- Implementation Level
 - detect **known** set of problems and security bugs
 - more automatic tools support available
 - target particular flaws
 - reviewing (auditing) software for flaws is reasonably well-known and well-documented
 - support for static and dynamic analysis
 - ranges from “how-to” for manual code reviewing to elaborate model checkers or compiler extensions
-

Static Security Testing

- Manual auditing
 - code has to support auditing
 - architectural overview
 - comments
 - functional summary for each method
 - OpenBSD is well known for good auditing process
 - 6 -12 members since 1996
 - comprehensive file-by-file analysis
 - multiple reviews by different people
 - search for bugs in general
 - proactive fixes: try to find and fix bugs before they are used in the wild
 - Microsoft also has intensive auditing processes
 - every piece of written code has to be reviewed by another developer
-

Static Security Testing

- Manual auditing
 - tedious and difficult task
 - some initiatives were less successful
- Sardonix (security portal)

“Reviewing old code is tedious and boring and no one wants to do it.”

- Crispin Cowan
- Linux Security Audit Project (LSAP)

Statistics for All Time

Lifespan		Rank		Page Views		D/l		Bugs		Support		Patches		Trkr		Tasks
1459 days		0 (0.00)		4,887		0		0 (0)		0 (0)		0 (0)		0 (0)		0 (0)

Static Security Testing

- Syntax checker
 - parse source code and check for functions known to introduce vulnerabilities
 - e.g., `strcpy()`, `strcat()`
 - also limited support for arguments (e.g., variable, static string)
 - only suitable as first basic check
 - cannot understand more complex relationships
 - no control flow or data flow analysis
 - examples / tools (all open source)
 - `flawfinder` (c/c++)
 - RATS (Rough Auditing Tool for Security, c/c++/perl/python/php)
 - ITS4 (c/c++)
-

Static Security Testing

```
int main(int argc, char **argv)
{
    if (argc > 1)
    {
        char buf[32];
        strcpy(buf, argv[1]);
        printf("this was the first argument: %s\n", buf);
    }
    ...
}
```

Static Security Testing

```
$ flawfinder base.c
```

```
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
```

```
Number of dangerous functions in C/C++ ruleset: 160
```

```
Examining base.c
```

```
base.c:13:  [4] (buffer) strcpy:
```

```
Does not check for buffer overflows when copying to destination.
```

```
Consider using strncpy or strncpy (warning, strncpy is easily misused).
```

```
base.c:12:  [2] (buffer) char:
```

```
Statically-sized arrays can be overflowed. Perform bounds checking,  
use functions that limit length, or ensure that the size is larger than  
the maximum possible length.
```

```
Hits = 2
```

Static Security Testing

```
$ rats base.c
```

```
Entries in c database: 336
```

```
...
```

```
Analyzing base.c
```

```
base.c:12: High: fixed size local buffer
```

```
Extra care should be taken to ensure that character arrays that are  
allocated on the stack are used safely. They are prime targets for buffer  
overflow attacks.
```

```
base.c:13: High: strcpy
```

```
Check to be sure that argument 2 passed to this function call will not copy  
more data than can be handled, resulting in a buffer overflow.
```

```
Total lines analyzed: 23
```

```
Total time 0.000237 seconds
```

```
97046 lines per second
```

Static Security Testing

```
int main(int argc, char **argv)
{
    if (argc > 1)
    {
        char buf[32];
        if (strlen(argv[1]) < 32)
        {
            strcpy(buf, argv[1]);
        }
        else
        {
            memcpy(buf, argv[1], 31);
            buf[31] = 0;
        }
        printf("this was the first argument: %s\n", buf);
    }
}
```

Static Security Testing

```
$ flawfinder good.c
```

```
...
```

```
good.c:14: [4] (buffer) strcpy:
```

Does not check for buffer overflows when copying to destination.

Consider using strncpy or strlcpy (warning, strncpy is easily misused).

```
good.c:12: [2] (buffer) char:
```

Statically-sized arrays can be overflowed. Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

```
good.c:17: [2] (buffer) memcpy:
```

Does not check for buffer overflows when copying to destination. Make sure destination can always hold the source data.

```
good.c:13: [1] (buffer) strlen:
```

Does not handle strings that are not \0-terminated (it could cause a crash if unprotected).

```
Hits = 4
```

Static Security Testing

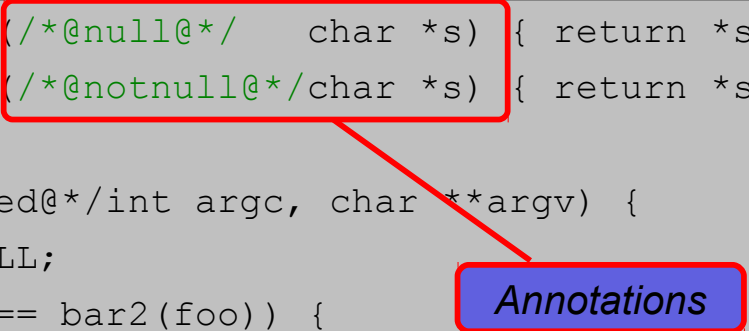
- Annotation-based systems
 - programmer uses annotations to specify properties in the source code (e.g., this value must not be NULL)
 - analysis tool checks source code to find possible violations
 - control flow and data flow analysis is performed
 - The problem is *undecidable* in general, therefore trade-off between *correctness* and *completeness*
 - examples / tools
 - SPlint
 - eau-claire
 - UNO (uninitialized vars, out-of-bounds access)
-

Static Security Testing

- Annotation-based systems: `SPlint`
 - tool for statically checking C programs for security vulnerabilities and coding mistakes
 - <http://www.splint.org>

```
static char bar1(/*@null@*/ char *s) { return *s; }
static char bar2(/*@nonnull@*/ char *s) { return *s; }

int main(/*@unused@*/ int argc, char **argv) {
    char *foo = NULL;
    if (bar1(foo) == bar2(foo)) {
        printf("we survived %s\n", argv[0]);
        // but we never do!!
    }
}
```



Static Security Testing

```
$ splint base.c
```

```
Splint 3.1.2 --- 13 May 2009
```

```
base.c: (in function bar1)
```

```
base.c:4:11: Dereference of possibly null pointer s: *s
```

A possibly null pointer is dereferenced. Value is either the result of a function which may return null (in which case, code should check it is not null), or a global, parameter or structure field declared with the null qualifier. (Use -nullderef to inhibit warning)

```
base.c:3:34: Storage s may become null
```

```
base.c: (in function main)
```

```
base.c:14:24: Null storage foo passed as non-null param: bar2 (foo)
```

A possibly null pointer is passed as a parameter corresponding to a formal parameter with no /*@null@*/ annotation. If NULL may be used for this parameter, add a /*@null@*/ annotation to the function parameter declaration. (Use -nullpass to inhibit warning)

```
base.c:13:14: Storage foo becomes null
```

```
Finished checking --- 2 code warnings
```

Static Security Testing

- Annotation-based systems: SPInt
 - Annotations for APIs

```
int main(int argc, char** argv) {  
    char *str;  
    if(argc < 2)  
        exit(0);  
    str = (char *)malloc(sizeof(char) * (strlen(argv[1])+1));  
    strcpy(str, argv[1]);  
    printf("Input String: %s \n", str);  
    return 1;  
}
```

*char *strcpy(char /*@nonnull@*/ str, char *src);*

*str = (char *)malloc(sizeof(char) * (strlen(argv[1])+1));*
strcpy(str, argv[1]);

malloc might return NULL
strcpy needs non-NULL dest param

Static Security Testing

```
$ splint mem.c
```

```
Splint 3.1.2 --- 13 May 2009
```

```
mem.c: (in function main)
```

```
mem.c:10:10: Possibly null storage str passed as non-null param: strcpy (str, ...)
```

A possibly null pointer is passed as a parameter corresponding to a formal parameter with no /*@null@*/ annotation. If NULL may be used for this parameter, add a /*@null@*/ annotation to the function parameter declaration. (Use -nullpass to inhibit warning)

```
mem.c:9:9: Storage str may become null
```

```
mem.c:12:12: Fresh storage str not released before return
```

A memory leak has been detected. Storage allocated locally is not released before the last reference to it is lost. (Use -mustfreefresh to inhibit warning)

```
mem.c:9:3: Fresh storage str created
```

```
Finished checking --- 2 code warnings
```

Static Security Testing

- Model checking
 - programmer specifies security properties that have to hold
 - models realized as state machines
 - statements in the program result in state transitions
 - certain states are considered insecure
 - usually, control flow and data flow analysis is performed
 - examples:
 - In Unix systems, *model checking* might verify that a program obeys the following rule: A setuid-root process should not execute an untrusted program without first dropping its root privilege.
 - race conditions
 - examples / tools
 - MOPS (an infrastructure for examining security properties of software)
-

Static Security Testing

- Meta-compilation
(Dawson Engler, Stanford University)
 - Programmer adds simple system-specific compiler extensions
 - these extensions check (or optimize) the code
 - flow-sensitive, inter-procedural analysis
 - not *sound*, but can detect many bugs
 - no annotations needed, instead states and state transitions
 - example extensions
 - system calls must check user pointers for validity before using them
 - disabled interrupts must be re-enabled
 - to avoid deadlock, do not call a blocking function with interrupts disabled
 - freed pointers must not be dereferenced / freed
-

Static Security Testing

- Meta-compilation, example
 - [<http://www.stanford.edu/~engler/p27-hallem.pdf>]
 - define state, state transitions and actions for certain states

```
state decl any_pointer v;

start:
    { kfree(v) } ==> v.freed;

v.freed:
    { *v }          ==> v.stop,
    { err("using %s after free!", mc_identifier(v)); }
    |
    { kfree(v) } ==> v.stop,
    { err("double free of %s!", mc_identifier(v)); };
```

Static Security Testing

- Meta-compilation, example

```
int contrived(int *p, int *w, int x) {  
    int *q;  
    if(x) {  
        kfree(w);  
        q = p;  
        p = 0;  
    }  
    if(!x)  
        return *w;  
    return *q;  
}
```

safe

using 'q' after free!

```
int contrived_caller (int *w, int x, int *p) {  
    kfree (p);  
    contrived (p, w, x);  
    return *w;  
}
```

using 'w' after free!

Static Security Testing

- Model checking versus Meta-compilation
 - “*Static Analysis versus Software Model Checking for Bug Finding*”
(Engler 2003, <http://www.springerlink.com/content/wx4ppjhwgt696dt8/fulltext.pdf>)
 - evaluated on 3 case studies over years
 - General perception
 - meta-compilation: easy to apply, but finds rather shallow bugs
 - model checking: harder, but strictly better once done
 - Example of Case Study: FLASH processor
 - . ccNUMA (Cache Coherent Non-Uniform Memory Access) with cache coherence protocols in software
 - code with many ad hoc correctness rules
 - `WAIT_FOR_DB_FULL` must precede `MISCBUS_READ_DB`
 - but they have a clear mapping to source code
 - easy to check with compiler
-

Static Security Testing

- Meta-compilation
 - scales
 - relatively precise
 - statically found 34 bugs (although code tested for 5 years)
 - however, many deeper properties are missed
 - Deeper properties
 - nodes never overflow their network queues
 - sharing list empty for dirty lines
 - nodes do not send messages to themselves
 - Perfect application for model checking
 - bugs depend on intricate series of low-probability events
 - self-contained system that generates its own events
-

Static Security Testing

- The (known) problem
 - writing model is hard
 - someone did it for a protocol similar to ccNUMA
 - several months effort
 - no bugs
 - auto-extract model from code
 - Result
 - 8 errors found
 - two deep errors, but 6 bugs found with static analysis as well.
 - Myth: model checking will find more bugs
 - in reality, 4x fewer
-

Static Security Testing

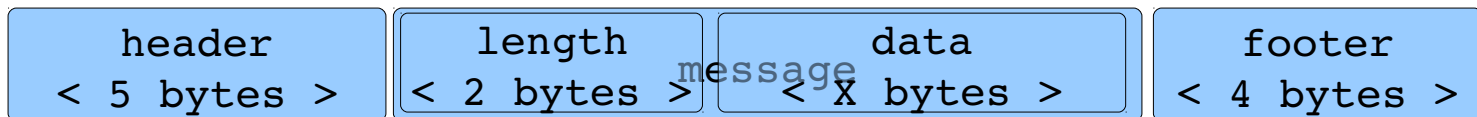
- However, model checking has advantages that seem hard for static analysis to match:
 - it can check the implications of code, rather than just surface-visible properties
 - properties such as deadlocks and routing loops involve invariants of objects across multiple processes.
 - Detecting such loops statically would require reasoning about the entire execution of the protocol, a difficult task.
 - it checks for actual errors, rather than having to reason about all the different ways the error could be caused
 - it gives much stronger correctness results — it would be surprising if code crashed after being model checked, whereas it is not at all if it crashes after being statically checked
-

Dynamic Security Testing

- Fuzz testing (*fuzzing*)
 - brute-force vulnerability detection
 - test programs with lots and lots of invalid or (semi-)random input
 - monitor program for crashes, dead-locks, etc.
 - particularly successful in finding protocol/file parsing errors
 - Originally an homework for an operating system class
 - program (the fuzz generator) to test commandline unix tools by providing them random data
 - Two groups failed, the third managed to crash 25-33% of the applications
 - The space to explore is very large
 - fuzzers use heuristics to find solutions in a reasonable time
 - often find only simple bugs
-

Dynamic Security Testing

- Very simple, yet very effective
 - Since 1988, applied practically to everything ... (or not?)
 - Books published on the topic
- Many tools available (especially for protocols and files)
 - model minimal protocol specification
 - fuzzer will randomize input bytes, but follow specification rules
 - OWASP JbroFuzzm, Peach, SPIKE,
 - Powerfuzzer, Protos, ...



Security Testing – (Pre-)Rollout

- Prepare code for release:

- remove debug code

```
if (checkPassword(user, passwd) ||  
    user == "test") {
```

- remove debugging information, symbols, etc.
 - *strip* binary
- remove sensitive information concerning possible weaknesses and untested code, disable debug output

```
if ($debug) {  
    print("user: $user\n");  
    print("auth: $auth_status\n");  
    ...  
}
```

- reset all security settings, remove test accounts, etc.
-

Security Testing – (Pre-)Rollout

- Penetration testing
 - a penetration test is the process of actively evaluating your information security measures
 - somehow similar to the course challenges
 - common procedure
 - analysis for design weaknesses, technical flaws and vulnerabilities
 - the results delivered comprehensively in a report (to executive, management, and technical audiences)
 - Why penetration testing: Why would you want it?
 - e.g., banks, gain and maintain certification (BS7799, NATO etc.)
 - assure your customers that you are security-aware
 - sink costs (yes, security bugs may cost you more)
-

Black-Box Vulnerability Scanners

- Tools designed to find known vulnerabilities in computer systems
 - Port scanners (Nmap)
 - Network scanners (Nessus, Retina, Saint, ..)
 - Web application scanners (loong list)
 - Pentesting support tools
 - Core Impact
 - Metasploit
-

Example: Testing Web Vuln. Scanners

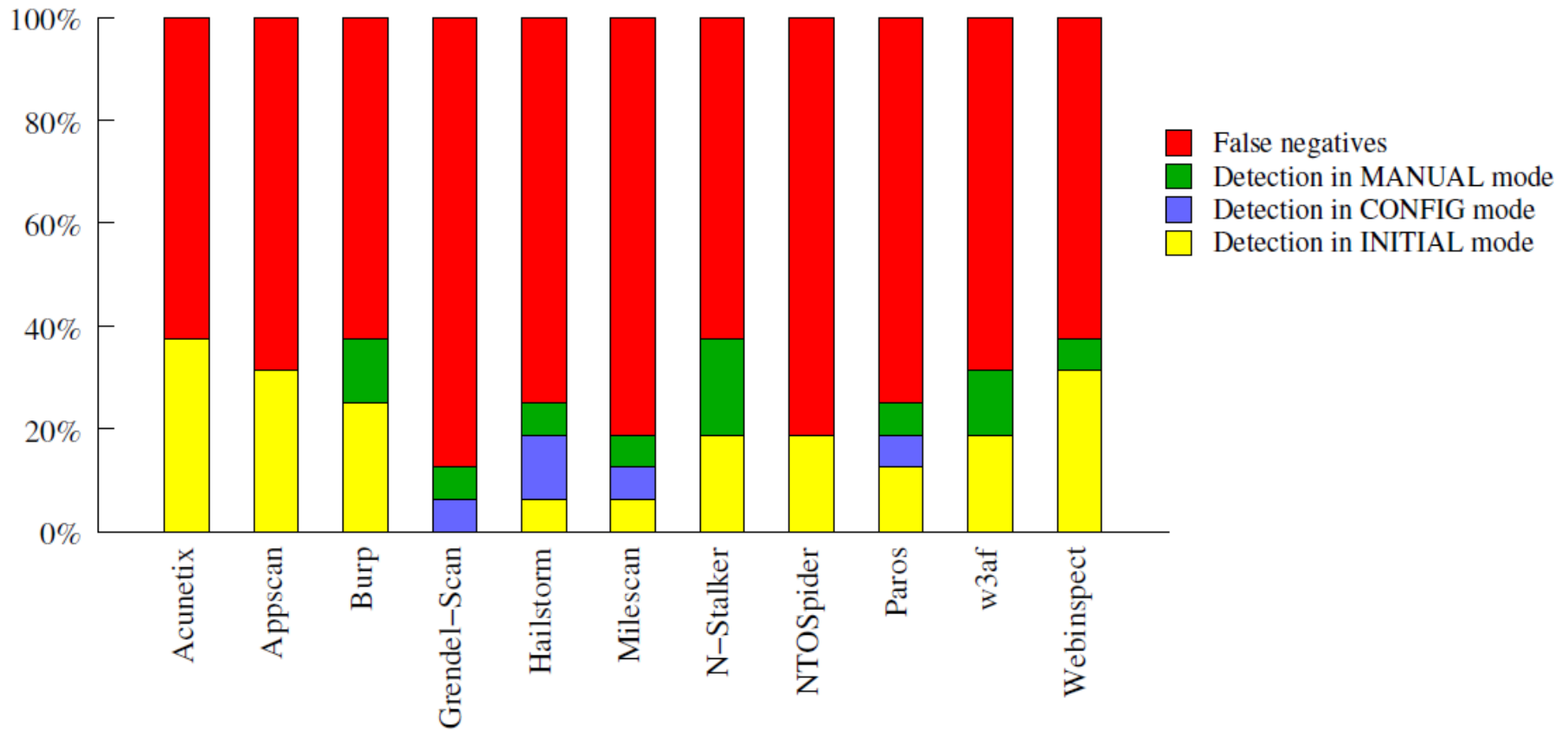
- Evaluation of eleven black-box web vulnerability scanners, both commercial and open-source.
- Tests are integrated in a realistic web application, containing many different vulnerabilities

Name	Version Used	License	Type	Price
Acunetix	6.1 Build 20090318	Commercial	Standalone	\$4,995-\$6,350
AppScan	7.8.0.0 iFix001 Build: 570 Security Rules Version 647	Commercial	Standalone	\$17,550-\$32,500
Burp	1.2	Commercial	Proxy	£125 (\$190.82)
Grendel-Scan	1.0	GPLv3	Standalone	N/A
Hailstorm	5.7 Build 3926	Commercial	Standalone	\$10,000
Milescan	1.4	Commercial	Proxy	\$495-\$1,495
N-Stalker	2009 - Build 7.0.0.207	Commercial	Standalone	\$899-\$6,299
NTOSpider	3.2.067	Commercial	Standalone	\$10,000
Paros	3.2.13	Clarified Artistic License	Proxy	N/A
w3af	1.0-rc2	GPLv2	Standalone	N/A
Webinspect	7.7.869.0	Commercial	Standalone	\$6,000-\$30,000

Example: Testing Web Vuln. Scanners

Name	Reflected XSS	Stored XSS	Reflected SQL Injection	Command-line Injection	File Inclusion	File Exposure	XSS via JavaScript	XSS via Flash
Acunetix	INITIAL	INITIAL	INITIAL	INITIAL	INITIAL	INITIAL	INITIAL	MANUAL
AppScan	INITIAL	INITIAL	INITIAL		INITIAL	INITIAL		
Burp	INITIAL	MANUAL	INITIAL			INITIAL		
Grendel-Scan	MANUAL		CONFIG					
Hailstorm	INITIAL	CONFIG	CONFIG					MANUAL
Milescan	INITIAL	MANUAL	CONFIG					
N-Stalker	INITIAL	MANUAL	MANUAL			INITIAL	INITIAL	MANUAL
NTOSpider	INITIAL	INITIAL	INITIAL					
Paros	INITIAL	INITIAL	CONFIG					MANUAL
w3af	INITIAL	MANUAL	INITIAL		INITIAL			MANUAL
Webinspect	INITIAL	INITIAL	INITIAL		INITIAL		INITIAL	MANUAL

Example: Testing Web Vuln. Scanners



Penetration Testing

- Different types of services
 - external penetration testing (traditional)
 - testing focuses on services and servers available from outside
 - internal security assessment
 - typically, testing performed on LAN, DMZ, network points
 - application security assessment
 - applications that may reveal sensitive information are tested
 - wireless / remote access assessment
 - e.g., wireless access points, configuration, range, etc.
 - telephony security assessment
 - e.g., mailbox deployment and security, PBX systems, etc.
 - social engineering (e.g., Kevin Mitnick)
 - e.g., passwd security, “intelligence” of users, etc.
-

Special Tips when choosing supplier

- *Who* should do the penetration testing?
 - do they have the necessary background?
 - technical sophistication, good knowledge of the field, literature, certification, etc.?
 - does the supplier employ ex-“hackers”?
 - beware of “consultants”
(let’s be a little critical and provocative ;-))
 - *Junior* = Person who has just started and who doesn’t necessarily know your domain better than you do ;-)
 - *Senior* = Person who manages, can present well, but has little technical knowledge ;-)
-

Special Tips when choosing supplier

- *Who* should *NOT* do penetration testing?
 - anyone who was not explicitly asked to do it
 - never pen-test a foreign/unknown system
 - you will (probably) be logged
 - It is an illegal activity
 - laws might be different (stricter) in other countries (i.e. where is the server/system you are targeting located?)
 - you might be held responsible for any damage you cause on a system
 - We want to encourage you to learn about security and writing secure code....
 - We don't want you to attend this course to end up in jail ;-)
-

Conclusion

- Testing
 - important part of regular software life-cycle
 - but also important to ensure a certain security standard
 - Important at design *and* implementation level
 - design: attack graphs, formal methods, manual reviews
 - implementation: static and dynamic techniques
 - Static techniques
 - code review, syntax checks, model checking, meta-compilation
 - Dynamic techniques
 - system call and library function interposition, profiling
-