

---

# Testing

## Internet Security [1] VU

Paolo Milani Comparetti, Christian Platzer,  
Gilbert Wondracek, **Markus Huber**, Edgar Weippl  
[inetsec@iseclab.org](mailto:inetsec@iseclab.org)

# News from the Lab

Int. Secure Systems Lab  
Vienna University of Technology

- *Challenge 6 – Stack Buffer Overflow*
  - continues two weeks
  - so far, solved by 9 people
  - *respect!*
  - DOMINIK "Coder Aceclown" CSAPAK Owned the stack first
    - congrats!!

# Outline

*Int. Secure Systems Lab  
Vienna University of Technology*

- So far ... offensive approach – exploit flaws
  - low level network attack
  - mid level protocol attacks
  - high level application attacks
- 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

Int. Secure Systems Lab  
Vienna University of Technology

- 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 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
  - white-box testing
    - testing all the implementation
    - path coverage considerations
    - faults of commission
    - find implementation flaws
    - but cannot guarantee that specifications are fulfilled
  - black-box testing
    - testing against specification
    - only concerned with input and output
    - faults of omissions
    - specification flaws are detected
    - but cannot guarantee that implementation is correct

# 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

- 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

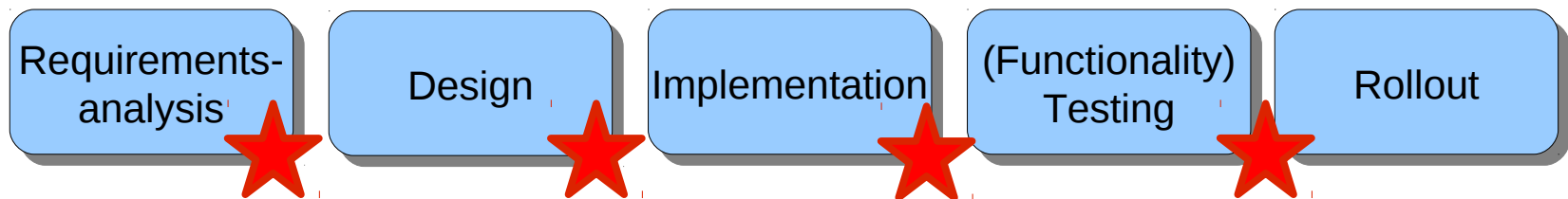
# Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- 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

- Many of the existing testing techniques can also be used to identify *security problems*
- Testing must happen at all different development cycle phases
  - test method depends on development phase
  - requirements analysis phase
  - design phase
  - implementation phase
  - (pre-)rollout phase



# Security Testing – Requirements Analysis Phase

Int. Secure Systems Lab  
Vienna University of Technology

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

# Security Testing – Design Phase

Int. Secure Systems Lab  
Vienna University of Technology

- 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
  - often used for small, *safety*-critical programs / program parts
    - e.g., control program for nuclear power plant
    - 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

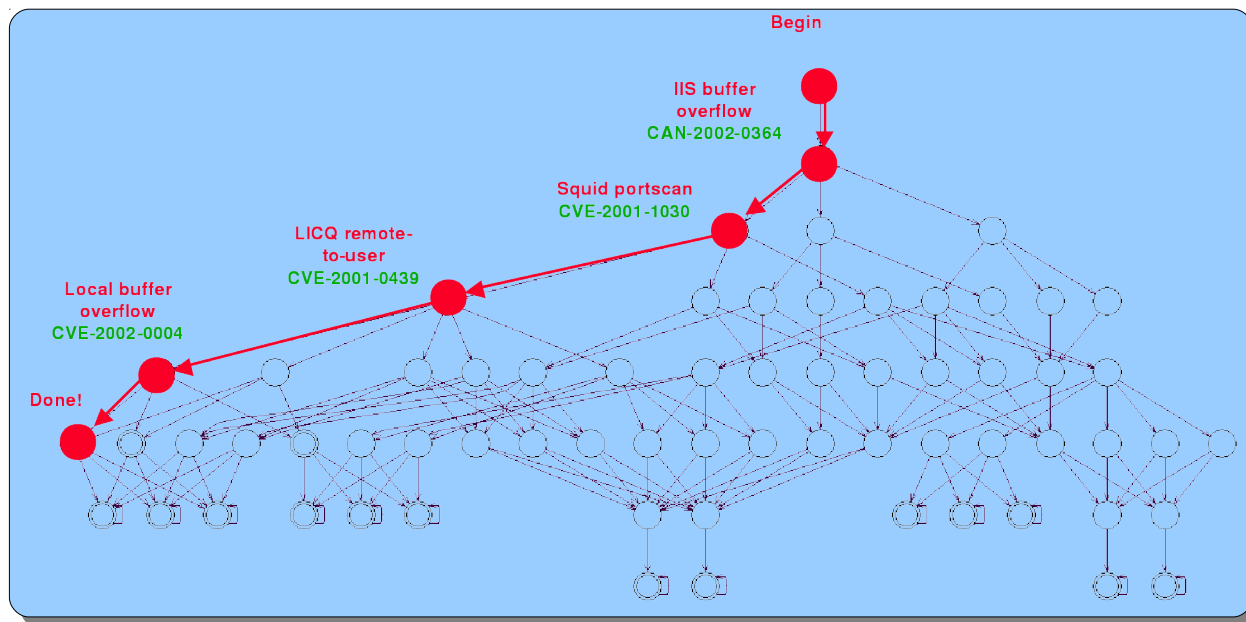
*Int. Secure Systems Lab  
Vienna University of Technology*

- Attack graph
  - given
    - a finite state model,  $M$ , of a network
    - a security property  $P$
  - an attack is an execution of  $M$  that violates  $P$
  - an attack graph is a set of attacks of  $M$
- Attack graph generation
  - done by hand
    - error prone and tedious
    - impractical for large systems
  - automatic generation
    - provide state description
    - transition rules

# Security Testing – Attack Graphs

Int. Secure Systems Lab  
Vienna University of Technology

Security property (LTL):  
 $G( intruder.privilege(host) < root )$



Somesh Jha, et al "Generation of Scenario Graphs Using Model Checking"

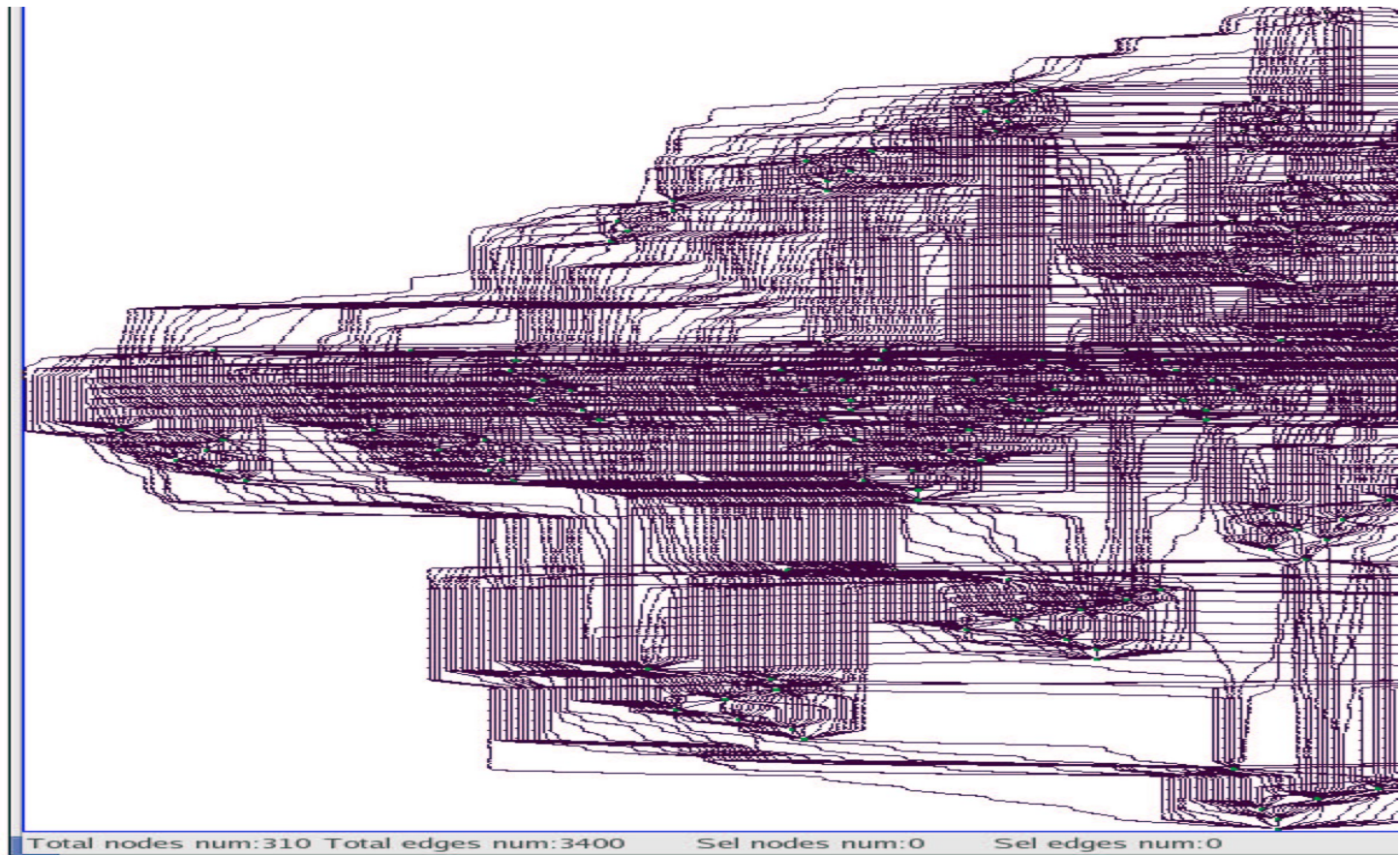


# Security Testing – Attack Graphs

*Int. Secure Systems Lab  
Vienna University of Technology*

P = Attacker gains root access to Host 1.

4 hosts  
30 actions  
310 nodes  
3400 edges



# Security Testing – Implementation

*Int. Secure Systems Lab  
Vienna University of Technology*

- Implementation Level
  - detect known set of problems and security bugs
  - more automatic tool 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

*Int. Secure Systems Lab  
Vienna University of Technology*

- Manual auditing
  - code has to support auditing
    - architectural overview
    - comments
    - functional summary for each method
  - OpenBSD is well know 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)

*Crispin Cowan: "Reviewing old code is tedious and boring and no one wants to do it."*

- Linux Security Audit Project (LSAP)

## Statistics for All Time

Lifespan	Rank	Page Views	D/1	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

*Int. Secure Systems Lab  
Vienna University of Technology*

- Syntax checker
  - parse source code and check for functions are known to introduce vulnerabilities
    - e.g., `strcpy()`, `strcat()` (as you saw in the buffer overflows lecture)
  - 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

- Syntax checker: `flawfinder`
  - find potential security flaws ("hits") in source code
  - limited to C/C++

```
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);
    }
}
```

*never !!*

*no !!*

*nein !!*

*non !!*

# Static Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

```
$ 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

*Int. Secure Systems Lab*  
*Vienna University of Technology*

```
$ 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);
    }
}
```

*Adding boundary check...*

# Static Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

```
$ flawfinder good.c
```

```
...
```

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

Does not check for buffer overflows when copying to destination.

Consider using strncpy or strncpy (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 unprote

```
Hits = 4
```

... even though it could do better here, all static tools have their limits ...

# Static Security Testing

Int. Secure Systems Lab  
Vienna University of Technology

- 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
  - problems are *undecidable* in general, therefore trade-off between *correctness* and *completeness*
    - *decidable*: there exists an algorithm that is guaranteed to return the correct answer in a *finite* amount of time
    - *undecidable*: Problem for which there *cannot* exist an algorithm that is guaranteed to terminate in all cases
  - examples / tools
    - SPInt
    - eau-claire
    - UNO (uninitialized vars, out-of-bounds access)

# Static Security Testing

- Annotation-based systems: SPInt
  - 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!!
    }
}
```

**Annotations**

# Static Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

```
$ 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: SPLint
  - Annotations for APIs

```
int main(int argc, char** argv) {  
    char *str; char *strcpy(char /*@nonnull@*/ str, char *src);  
    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;  
}
```

*malloc might return NULL  
strcpy needs non-NULL dest param*

# Static Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

```
$ 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

Int. Secure Systems Lab  
Vienna University of Technology

- 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
  - example:
    - 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
    - creating a secure chroot jail
  - examples / tools
    - MOPS (an infrastructure for examining security properties of software)

# Static Security Testing

- Model checking, example

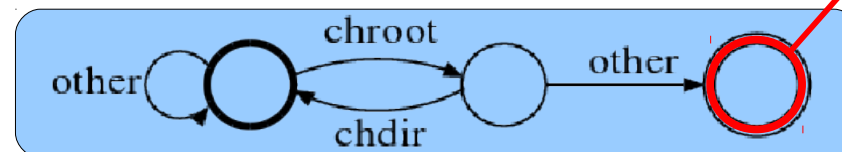
- [http://www.cs.ucdavis.edu/~hchen/paper/ccs02.pdf]
- suppose a process uses the **chroot** system call to confine its access to a sub filesystem. In this case, the process should immediately call **chdir("/")** to change its working directory to the root of the sub filesystem.

`chdir("/");` missing

```
// Here the current directory is "/var/ftp"  
chroot("/var/ftp/pub");
```

```
filename = read_from_network();  
// filename could be "../..etc/passwd"  
fd = open(filename, O_RDONLY);
```

*insecure state!*



# Static Security Testing

Int. Secure Systems Lab  
Vienna University of Technology

- 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

*Int. Secure Systems Lab  
Vienna University of Technology*

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

safe

using 'q' after free!

using 'w' after free!

# Static Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- Model checking versus Meta-compilation (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
- FLASH processor: ccNUMA (Cache Coherent Non-Uniform Memory Access) with cache coherence protocols in software
  - 1 bug deadlocks entire machine
  - 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

*Int. Secure Systems Lab  
Vienna University of Technology*

- 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

*Int. Secure Systems Lab  
Vienna University of Technology*

- The (known) problem
  - writing model is hard
  - someone did it for a similar protocol than ccNUMA
    - several months effort
    - no bugs
  - auto-extract model from code
- Result
  - 8 errors
  - 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

Int. Secure Systems Lab  
Vienna University of Technology

- Where meta-compilation is superior

	Static analysis	Model checking
	Compile → Check	Run/Simulate → Check
Don't understand?	So what	Problem
Can't run?	So what	Can't play
Coverage?	All paths! All paths!	Executed paths
First question:	"How big is code?"	"What does it do?"
Time:	Hours	Weeks
Bug counts	100-1000s	0-10s
Big code:	10MLOC	10K
No results?	Surprised	Less surprised

# Static Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- Where model checking is superior
- Subtle errors
  - run code, so can check its implications
  - static analysis better at checking properties in code
  - model checking better at checking properties implied by code
- Difference
  - static analysis detects ways to cause error
  - model checking checks for the error itself

# Static Web Script Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- Our tool: `pixy`
  - developed by Nenad Jovanovic
  - analyzes PHP source code and automatically detects XSS/SQLi vulnerabilities
  - web interface: <http://pixybox.iseclab.org/pixy/>
  - client version downloadable

## SHORT DEMO

# Dynamic Security Testing

Int. Secure Systems Lab  
Vienna University of Technology

- *Run-time* checking between operating system and program
  - intercept and check system calls
- *Run-time* checking between libraries and program
  - intercept and check library functions
  - often used to detect memory problems
    - interception of `malloc()` and `free()` calls
    - emulation of heap behavior and code instrumentation
    - `purify`, `valgrind`
  - also support for *buffer overflow* detection
    - `libsafe`

# Dynamic Security Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- Profiling
  - record the dynamic behavior of applications with respect to interesting properties
- Obviously interesting to tune performance
  - gprof
- But also useful for improving security
  - sequences of system calls
  - system call arguments
  - same for function calls

# Dynamic Security Testing

Int. Secure Systems Lab  
Vienna University of Technology

- Fuzz testing (*fuzzing*)
  - brute-force vulnerability detection
  - penetrate program with lots and lots of (semi-)random input
  - monitor program for crashes, dead-locks, etc.
  - particularly successful in finding protocol/file parsing errors
- Tools
  - model minimal protocol specification



- fuzzer will randomize input bytes, but follow specification rules
- OWASP JbroFuzzm,
- SPIKE,
- Powerfuzzer

# 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

*Int. Secure Systems Lab  
Vienna University of Technology*

- Penetration testing
  - a penetration test is the process of actively evaluating your information security measures
  - somewhat similar to Inetsec 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)

# Penetration Testing

Int. Secure Systems Lab  
Vienna University of Technology

- OK... so *how* do people do it?
  - general tool support available
    - Nessus
    - ISS Internet Scanner
    - nmap
  - tools available that can test a particular protocol
    - Whisker, w3af (web)
    - Internet Security Systems (ISS) Database scanner
  - Special penetration testing suites
    - Backtrack Linux

# Penetration Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- 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

# Penetration Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- Different types of services
  - 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., piggybacking, phone calls, etc.

# Limitations of Penetration Testing

*Int. Secure Systems Lab  
Vienna University of Technology*

- Permission to attack
  - Client defines targets beforehand
  - Only certain systems allowed
  - Tests on demo systems
- Actual penetration testing vs. report writing
  - Client pays for report
  - Report writing takes a lot time
  - “Pretty” reports valued more than sophistication of exploits

# Special Tips when choosing supplier

Int. Secure Systems Lab  
Vienna University of Technology

- *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

Int. Secure Systems Lab  
Vienna University of Technology

- *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
    - 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 (e.g. SQL injection “'or drop table foobar--”, DoS)
- Some students were inspired by the command- and SQL-injection lectures
  - thanks for the funny links & screenshots we received
    - e.g. derstandard.at
  - BUT: 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

# Outlook

*Int. Secure Systems Lab  
Vienna University of Technology*

- Next week, we'll take a look at online social networks
  - Security/Privacy issues with Facebook & Co.
  - Recent attack vectors
  - Social Engineering via Social Networks
  - New unpublished findings
- The last lecture :-)