Language-Based Security

Lecture 3

CS4105 - Software Security

Q2 / 2015 – 2016

Eelco Visser*

TU Delft

* heavily borrowing from various sources; see last slide for list of sources

Software Security

- Software security is a subset of software **reliability**
- Security is about **protection of assets**
  - specified in terms of security requirements
  - confidentiality, integrity, availability, accountability
- Security is realized through **security mechanisms**
  - authentication, authorization, auditing
- **Threat modeling** used to identify threats against security
  - trust boundaries, attack surface
  - attack taxonomies (STRIDE), attack trees, attack libraries
- **Principles** of secure software design
  - best practices to avoid known pitfalls

TU Delft

Secure Software Design Approaches

Prevention

- Eliminate software defects entirely

Mitigation

- Reduce harm from exploitation of unknown defects

Detection and recovery

- Identify and understand an attack and undo damage

TU Delft

Preventing Security Bugs

Apply best practices in design of a software system

- Provides guidelines for avoiding vulnerabilities

No guarantees

- forget to check array bounds …
- forget to require authentication on a user interface with sensitive data …

Depends on

- programmer discipline
- code reviews
- bug finding tools
- …

Can we prevent security bugs altogether?

TU Delft

Safety Policies

Control flow safety

- Program should never execute jump or call to random location. Calls should be to valid function entry points and all returns to the location from which the function was called.

Memory safety

- The program should not access random places in memory but only valid locations

Stack safety

- For stack-based runtime architectures, the runtime stack should be preserved across function calls.

How are these properties violated?

- How can we ensure programs have these properties?

TUDelft

Memory Safety

A program execution is memory safe if

- It only creates valid pointers through standard means
- Only uses a pointer to access memory that belongs to that pointer

Combines temporal safety and spatial safety

Spatial Safety

Access only to memory that pointer owns

View pointer as triple (p, b, e)

- p is the actual pointer
- b is the based of the memory region it may access
- e is the extent (bounds of that region)

Access allowed iff

- `b <= p <= e - sizeof(typeof(p))`

Allowed operations

- Pointer arithmetic increments p, leaves b and e alone
- Using &: e determined by size of original type

Temporal Safety

A **temporal safety violation** occurs when trying to access undefined memory

- Spatial safety assures it was to a legal region

- Temporal safety assures that region is still in play

Memory region is defined or undefined

Undefined memory is
- unallocated,
- uninitialized, or
- deallocated

Checking Safety Properties

When is this statement safe?

```
a[i] = c
```

- a is an array (type safety)
- i is within bounds of the array a (memory safety)
- the value of c fits in cells of the array (type safety)

Who is responsible for ensuring this?

- *Programmer* (discipline): unsafe (no guarantees)
- *Static analysis*: safe

Language-Based Security

Prevent vulnerabilities entirely by

- Building safety guarantees into the programming language
- Type system: Static analysis applied by compiler
- Dynamic languages: Safe runtime

Slogan: “**Well-typed programs don’t go wrong**”

- Note: does not defend against all threats, just a particular category of (security) bugs

TU Delft

DEFENSIVE PROGRAMMING

strcpy (and friends)

```
SYNOPSIS
     #include <string.h>

     char *
     strcpy(char *restrict dst, const char *restrict src);

     char *
     strncpy(char *restrict dst, const char *restrict src, size_t n);

DESCRIPTION
     The stpcpy() and strcpy() functions copy the string src to dst (including
     the terminating `\0' character).

     The stpncpy() and strncpy() functions copy at most n characters from src
     into dst.  If src is less than n characters long, the remainder of dst is
     filled with `\0' characters.  Otherwise, dst is not terminated.

     The source and destination strings should not overlap, as the behavior is
     undefined.
```

Buffer Overflow Vulnerability

```
void main(int argc, char **argv) {
  char program_name[256];
  strcpy(program_name, argv[0]);
  f(program_name);
}
```

```
void strcpy(char *dst, char *src) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++] != '\0')
}
```

Problem: argv[0] may not fit in program_name

Violates: memory safety / stack safety

Prevention: Provide Upper Bound

```
void main(int argc, char **argv) {
  char program_name[256];
  strncpy(program_name, argv[0], 256);
  f(program_name);
}
```

```
void strncpy(char *dst, char *src, int n) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++] != '\0' && i < n)
}
```

Problem: String program_name may not be null terminated.

Violates: memory safety / stack safety

Prevention: Guarantee String Terminator

```
char *copy(char *s) {
  char buffer[BUF_SIZE];
  strncpy(buffer, s, BUF_SIZE-1);
  buffer[BUF_SIZE-1] = '\0';
  return buffer;
}
```

Problem: This program returns a pointer to *local* memory.

Violates: temporal safety / stack safety

Prevention: Allocate Buffer on the Heap

```
char *copy(char *s) {
  char *buffer = (char *)malloc(BUF_SIZE);
  if(buffer == NULL) error("…");
  strncpy(buffer, s, BUF_SIZE-1);
  buffer[BUF_SIZE-1]= '\0';
  return buffer;
}
```

Prevented: buffer overflow

Problem: may truncate string

Potential problem: memory leak

Prevention: Tracking Buffer Sizes

```
char *copy(char *s, int size) {
  char *buffer = (char *)malloc(size);
  if(buffer == NULL) error("…");
  strncpy(buffer, s, size+1);
  buffer[size]= '\0';
  return buffer;
}
```

if length of s is (smaller than) size then target buffer right size & string is not truncated

Problem: tracking of size may go wrong

Buffer API: Size Tracking Built-In

```
typedef struct {
  char* ptr;
  int bufsize;
} buffer;
buffer *alloc_buf(int size) {
  buffer *buf = (buffer *)malloc(sizeof(buffer));
  buf->bufsize = size;
  buf->ptr = (char *)malloc(buf->bufsize);
  return buf;
}
buffer *copy(buffer *src) {
  buffer *dst = alloc_buf(src->bufsize);
  strncpy(dst->ptr, src->ptr, dst->bufsize);
  dst->ptr[dst->bufsize-1] = '\0';
  return dst;
}
```

Invariant: bufsize is the size of the buffer assigned to ptr

API maintains invariant

Problem: C does not enforce API

`buff.ptr[buf.bufsize]`

MITIGATING OVERFLOWS

What makes buffer overflow possible?

Access to arbitrary memory location through pointer
- access of memory outside of boundaries
- pointer arithmetic (p = p + 10; *p = 3;)
- access of memory after it is freed

Execution of data as code
- overflowing code to stack, then jump to it
- function pointers

Leaky abstractions
- invariants are not kept / enforced
- function makes assumptions on arguments, caller violates assumption

TU Delft

Secure Coding

Robust Programming

- Avoid depending on anyone else around you
- If someone does something unexpected, you won't crash
- Minimize trust
- Each module pessimistically checks its assumed preconditions on outside callers
- Even if you know clients will not send a NULL pointer, better throw an exception than run malicious code
- Read: Robust Programming by Matt Bishop http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Use safe string functions

- Traditional string library functions assume target buffers have sufficient length
- Safe versions check the destination length

TU Delft

Secure Coding

- Don’t forget NUL terminator
- Understand pointer arithmetic
- Defend against dangling pointers
- Manage memory properly
- Use safe string library
- Favor safe libraries; libraries encapsulate well-thought-out design
- Use a safe collector: challenge heap-based overflows by making addresses returned by malloc unpredictable

TU Delft

Architectural Defenses

It is hard to get C programs right

- Measures in execution environment to mitigate their effect

Stack canaries

- An extra value on stack frame to check that frame was not overwritten

Data execution protection (DEP)

- Make stack and heap non-executable
- To prevent from executing code injected by attacker

Address space layout randomization (ASLR)

- To prevent injecting addresses that point to known library code
- To make guessing the location of the return address harder

LANGUAGE-BASED SECURITY

Can we do better?

Abstractions

Abstractions are crucial to **reduce complexity**

- Reducing complexity avoids bugs, and hence security problems

Abstractions **enable/provide security** (access control)

- Access to files (bits on the hard disk) by users & processes
- Provided the abstractions are rigorously enforced

But .. **some abstractions may be broken**

- Stack overflows break the procedure call mechanism
- Uninitialised virtual memory may leak information
- Timing attacks may reveal the virtual memory abstraction

Programming Languages and Security

Programming language can help security

By making certain security bugs less likely or impossible

- Impose some discipline or restrictions on the programmer
- Offer and/or enforce some abstractions to the programmer
- No buffer overflows possible in any decent language

By offering useful building blocks for security functionality

- Language support or APIs for access control

By making assurance of security easier (meta-property)

- Code review only of public interface
- This may allow security guarantees in the presence of untrusted, possibly malicious, code

TU Delft

Example: Arrays in Java

Declaration

- `int[] anArray;`

Memory allocation

- `anArray = new int[10];`

Initialization / assignment

- `anArray[0] = 100;`

Access / Indexing (zero-based)

- `System.out.println(anArray[0]);`

What is different from arrays in C?

Array *Abstraction* in Java

Type-specific memory allocation

- `anArray = new int[10]; // Java`
- `anArray = (int*)malloc(sizeof(int), 10); // C`

Array length is fixed when allocated

- `anArray.length`
- automatic buffer size tracking

Bounds checking

- `anArray[10] => IndexOutOfBoundsException // Java`
- `anArray[10] => undefined (just read memory) // C`
- throws exception when accessing array out of bounds

Array value is not a pointer

- `anArray + 10 => type error // Java`
- `*(anArray + 10) => peek into memory // C`

Memory Safety

A programming language is memory-safe if it guarantees that a program can

Never reference unallocated or de-allocated memory

- No segmentation faults at runtime

Never reference uninitialised memory

- => We could switch off OS access control to memory
- => We don’t have to zero out memory before de-allocating it to avoid information leaks
- Assuming there are no bugs in our execution engine …

TU Delft

Java Security

Array bounds checking

- Store dimensions of array with data
- Check that array is not accessed out of bounds
- Throw exception on out of bounds exception

Bytecode verification

- Ensure basic properties of memory, control-flow, and type safety

Security manager

- Enforce higher-level safety policies such as restricted I/O

Types

Types assert certain invariant properties

- through annotations on program elements
- ‘This variable will always hold an integer’
- ‘This variable will always refer to an object of class X (or one of its subclasses)’
- ‘This array will never store more than 10 items’

Type checking verifies the assertions

- A language is **type sound** if the assertions are guaranteed to hold at run-time
- aka **type safety** or **strong typing**

Type Information

Function argument is always of type ArithC

```
sealed abstract class ArithC
case class NumC (num:Int)                  extends ArithC
case class PlusC(l:ArithC,r:ArithC) extends ArithC
case class MultC(l:ArithC,r:ArithC) extends ArithC

object Interp {
  def interp(e:ArithC): Int = e match {
    case NumC(n)     => n
    case PlusC(l,r) => interp(l) + interp(r)
    case MultC(l,r) => interp(l) * interp(r)
  }
}
```

Constructor argument always has type ArithC,

Function always returns value of type Int

Type Safety

A programming language is **type-safe** if it can guarantee that

- programs that pass the type-checker
- can only manipulate data in ways allowed by their types

Program cannot 'go wrong', e.g. cannot

- add booleans,
- dereference integers,
- multiply references
- …

For OO languages

- no "Method not found" errors at runtime

Type Safe Languages

Memory-safe, typed and type sound languages

- Java, C#,
- functional languages like ML, Haskell, Clean, F#
- Some (e.g. Java and C#) still have unsafe features

Memory-safe, untyped languages

- Lisp,Prolog, many interpreted languages

Memory-unsafe, typed, type-unsafe languages

- C, C++, Pascal
- Not type sound
- Using pointer arithmetic in C, you can do anything you want and break any assertion made by the type system – breaking type soundness

Penalty of Safety

Performance vs safety

- Enforcing safety policies dynamically has performance penalty
- Typical enforcement of type safety is expensive
- New languages aiming to provide similar features to C/C++ while remaining type safe
- Google’s Go
- Mozilla’ Rust
- Apple’s Swift

TU Delft

Remaining Buffer Overflow Issues in Java/C#

Buffer overflows can still exist

- In native code
- For C#, in code blocks declared as unsafe
- Through bugs in the Virtual Machine (VM) implementation, which is typically written in C++ ….
- Through bugs in the implementation of the type checker, or worse, bugs in the type system (unsoundness)

How do we know type system is sound?

Representation independence (for booleans)

- It does not matter if we represent true as 0 and false as 1 (or FF), or vice versa
- If we execute a given program with either representation in the result will be the same
- One could test this, or try to prove it. (how?)
- Similar properties should hold for all datatypes.

How do we know type system is sound?

Prove the equivalence of

- A typed operational semantics, which records and checks type information at runtime
- An untyped operational semantics, which does not
- For all well-typed programs

Or, in other words, prove the equivalence of

- A defensive execution engine, which records and checks type information, and
- An ‘offensive’ execution engine which does not
- For any program that passes the type checker

- People have formalised the semantics and type system of e.g. Java using theorem provers (Coq, Isabelle/HOL) to then prove such results

TUDelft

Other Language-Based Guarantees

Visibility

- public, private, etc
- e.g. private fields not accessible from outside a class

Constants/immutability

- of primitive values
- in Java: `final int i = 5;`
- in C(++) : `const int BUF_SIZE = 128;`
- Beware: meaning of const gets confusing for C(++) pointers and objects!
- In Java, for example String objects are constants
- Scala provides a stronger distinction between mutable and immutable objects

TU Delft

Ongoing Evolution of Type Systems

Many ways to enrich type systems further

Distinguishing non-null and possibly-null types

- `public nonNull String hello;`
- alias control
- improve efficiency
- prevent bugs (namely NullPointerExceptions)
- at least catching them earlier, at compile time
- restrict possible interferences between modules due to aliasing

Information flow

- imposing restrictions on the way tainted information flows through a program

FORMAT STRING ATTACK

```
printf(user)
```

Exploiting Format String Vulnerabilities
https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Format Strings

```
SYNOPSIS
  #include <stdio.h>

  int
  printf(const char * restrict format, ...);

  int
  fprintf(FILE * restrict stream, const char * restrict format, …);

EXAMPLES

  To print a date and time in the form ``Sunday, July 3, 10:02'',
  where weekday and month are pointers to strings:

  #include <stdio.h>
  fprintf(stdout, '%s, %s %d, %.2d:%.2d\n”, weekday, month, day, hour, min);
```

match format parameters to arguments

Format String is Interpreted at Run-Time

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

% character is escape character;
identifies hole in the string to be
filled with content from (next) argument

stack top
. . .
<&a>
<a>
<i>
A
. . .
stack bottom

where:

A	address of the format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

Vulnerability: User-Provided Format String

```
char tmpbuf[512];
snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```

`user` variable contains user provided input

% in input interpreted as format parameter

indirect usage hard to detect

```
int Error (char *fmt, ...);
…
int someotherfunc (char *user)
{
  …
  Error (user);
  …
}
…
```

What can attacker do?

Crashing the program

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

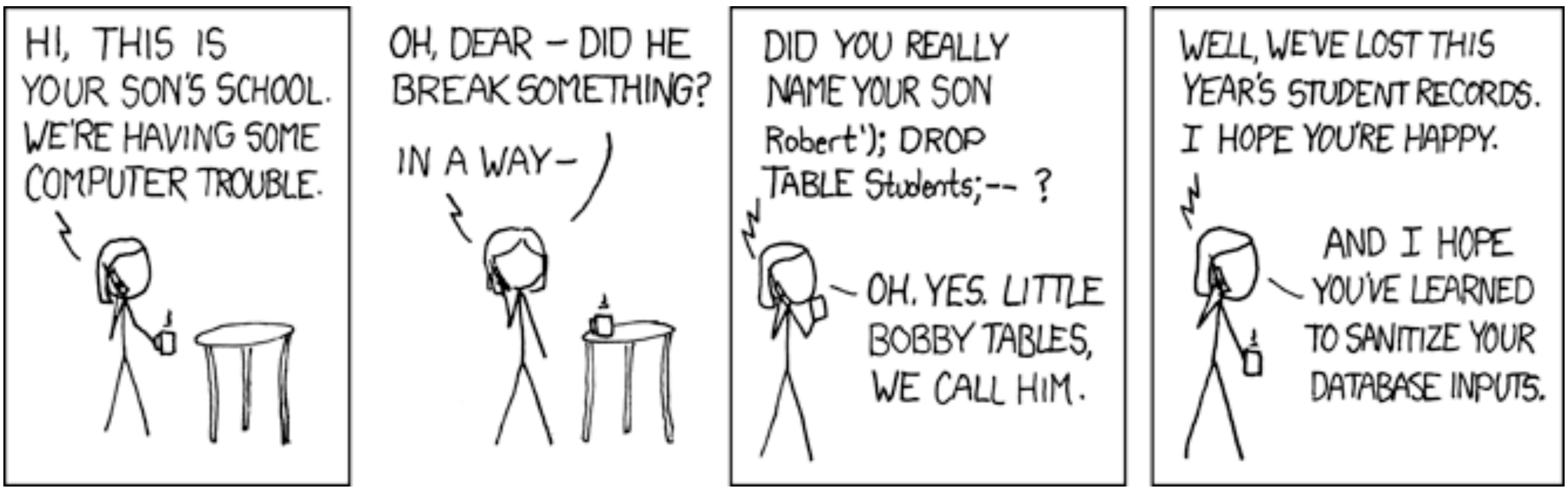
Viewing the stack

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

Dump memory from 0x08480110 until a NUL byte is reached

```
printf ("\x10\x01\x48\x08_%08x.%08x.%08x.%08x.%08x|%s|");
```

Overwriting arbitrary memory using %n format parameter


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

TU Delft

Problem: Interpreting User Input

C format strings

- printf(user)
- unchecked format parameters

JavaScript eval

- `eval(user);`
- Interpreting user generated input as JavaScript code
- (much more powerful than C format strings)

SQL queries

- ```
  select * from User
  where name=$name and password=$pass
  ```

Problems

- unintentionally interpreting user input
  ```

Lectures

Week 4: Vulnerabilities in web applications

- SQL injection
- Cross-site scripting
- Cross-site request forgery
- …

Week 5: Preventing web security bugs

- Programmer discipline: validating input
- Safe language mechanisms

- Lecture 1: What is Software
  - Eelco Visser in Delft
- Lecture 2: Memory-Based A
  - Sandro Etalle in Twent
- Lecture 3: Language-Based
  - Eelco Visser in Delft
- Lecture 4: Vulnerabilities in Web Applications (Dec 2)
  - Sandro Etalle in Twente
- Lecture 5: Language-based Security for the Web (Dec 9)
  - Danny Groenewegen, Mark Jansen in Delft
- Lecture 6: Information Flow and Access Control (Dec 16)
  - Eelco Visser in Delft
- Lecture 7: Security Testing (Jan 6)
  - Eelco Visser in Delft

Assignment D
Security Design and Analysis

D1: Threat Modeling

- Select an existing software system or imagine one
- Describe its functional design using standard modeling techniques
  - class diagrams
  - data-flow diagrams
  - use cases
- Apply threat modeling to the design
  - abuses cases
  - attack trees

D2: Threat Model Peer Review

D3: Designing Security Policies

- Formulate a security design, including authentication, authorization, and auditing policies for the D1 system and argue why your design is safe

D4: Security Policies Peer Review

Assignment I: Security Bugs and Language-Based Security

I1: Buffer Overflows

- Construct an attack by exploiting a buffer overflow vulnerability

I2: Web Security

- Implement a small web application with vanilla use of a web programming language / framework
- Examine security vulnerabilities in the result
- What do you need to do to prevent these bugs?
- Examine the counter measures in a WebDSL implementation of the same application

I3: Safety by Construction

- Implement a translation from a high-level language to a low-level language that ensures safety properties

Assignments & Deadlines

Assignment	Weight		Due
Lab	50.0 of total 100.0		
D1: Threat Modeling	5.0 of total 10.0	Dec 4	Nov 27
I1: Buffer Overflow	10.0 of total 50.0	Dec 11	Dec 4
D2: Threat Model Review	5.0 of total 10.0	Dec 18	Dec 11
I2: Web Security	10.0 of total 50.0	Jan 4	Dec 18
D3: Security Policies	5.0 of total 10.0	Jan 8	Jan 8
I3: Safety by Construction	10.0 of total 50.0	Jan 15	Jan 15
D4: Review Security Policies	5.0 of total 10.0	Jan 22	Jan 22

proposal

DSyS Lectures on Advances in Security Science and – Technology

04 december 2015 | 11:00 - 13:00

plaats: Faculty of TPM, TU Delft

door *Webredactie*

TU Delft is doing a lot of research related to security technology and – science. On Dec. 4th a mini-symposium will be held at the faculty of TPM in which 5 speakers give short presentations on a variety of security topics, as shown in the programme.

Program:

11.00 – 11.20: Important research topics in security science, Prof. Pieter van Gelder, TPM

- 11.20 – 11.40: Physical terror vs. cyber terror, Mr. Johan de Wit, TPM and Siemens
- 11.40 – 12.00: Social network analyses to quantify threat levels, Dr. Ana Barros, TNO and NLDA
- 12.00 – 12.20: Short lunch break
- 12.20 – 12.40: Predictive policing, Dr. Marielle den Hengst, TBM and Police Academy
- 12.40 – 13.00: The power of sensor technology, Prof. Alexander Yarovoy, EWI

We kindly ask you to register with Roy Weidmann for reasons of logistics.

Early next year, a larger symposium on this theme will be organized, including speakers from government, academia and businesses, for which you will receive an invitation in due course.

TU Delft

Sources

These slides are based on material from

- “Language-Based Security” by Dexter Kozen in Mathematical Foundations of Computer Science 1999
- “What is memory safety?” by Michael Hicks (blog)
- “What is type safety?” by Michael Hicks (blog)
- “Language-Based Security: Safety” by Erik Poll (slides)
- “Lecture 3, CIS/TCOM 551, Computer and Network Security” by Steve Zdancewic (slides)
- “Secure Programming with Static Analysis” by Brian Chess and Jacob West, Addison-Wesley, 2007
- “Exploiting Format String Vulnerabilities” by scrut / team teso, 2001
- Bobby Tables by xkcd

TUDelft