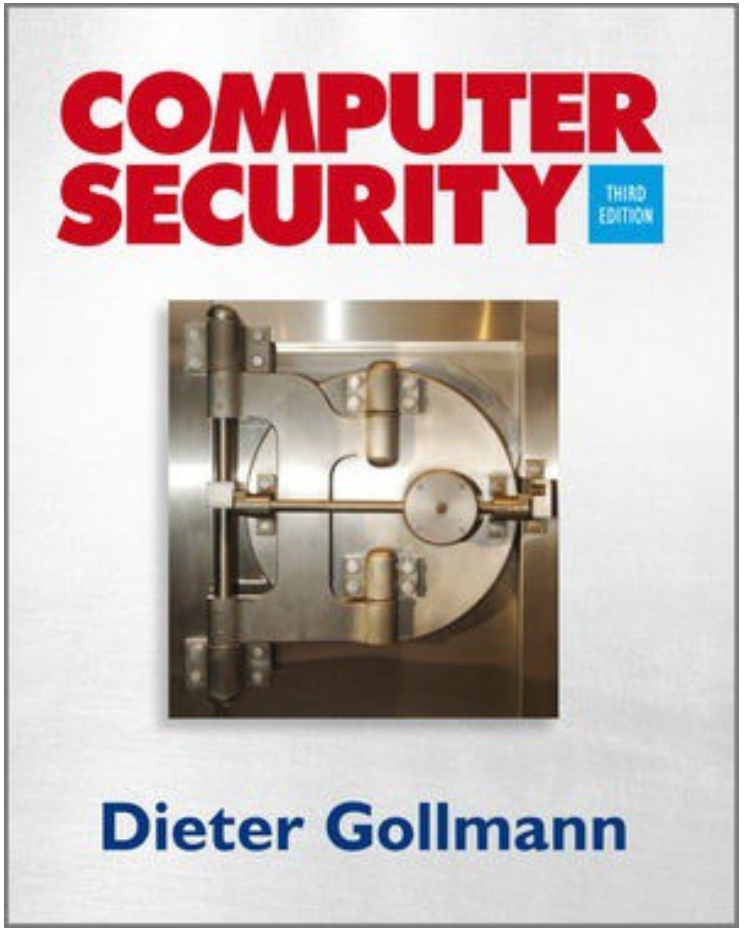
Computer Security 3e

Dieter Gollmann



Chapter 10: Software Security

Defences

Prevention – Hardware

- Hardware features can stop buffer overflow attacks from overwrite control information.
- For example, a secure return address stack (SRAS) could protect the return address.
- Separate register for the return address in Intel's Itanium processor.
- With protection mechanisms at the hardware layer there is no need to rewrite or recompile programs; only some processor instructions have to be modified.
- Drawback: existing software, e.g. code that uses multi-threading, may work no longer.

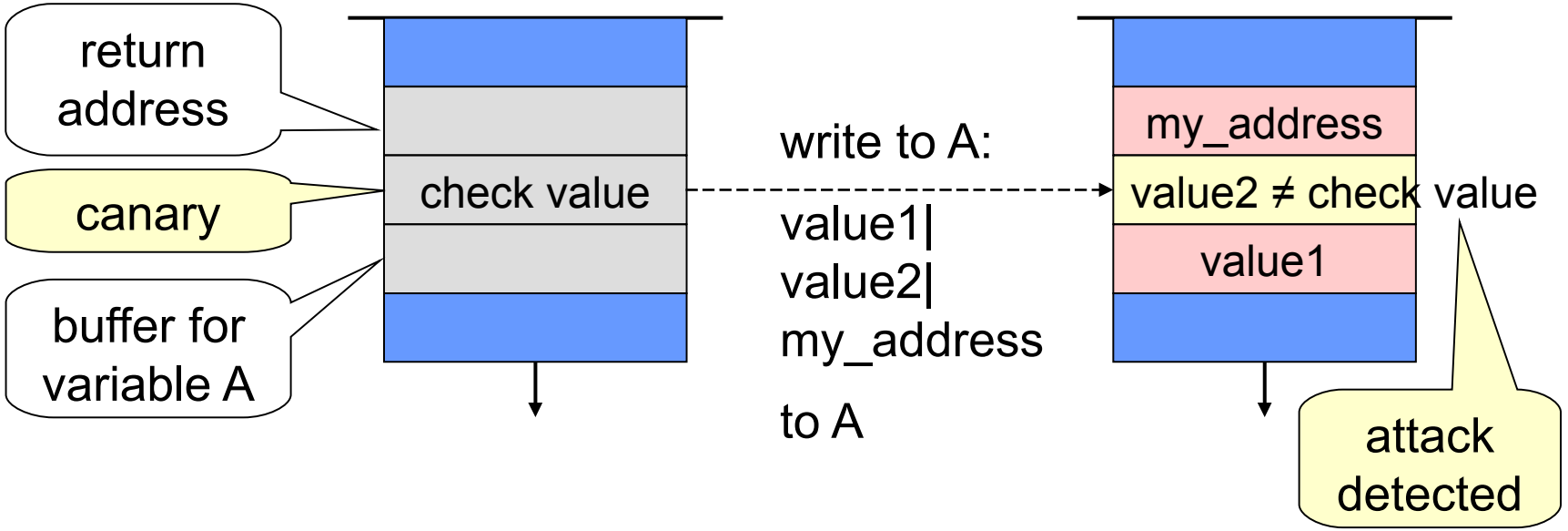
Prevention – Non-executable Stack

- Stops attack code from being executed from the stack.
- Memory management unit configured to disable code execution on the stack.
- Not trivial to implement if existing O/S routines are executing code on the stack.
- General issue – backwards compatibility: security measures may break existing code.
- Attackers may find ways of circumventing this protection mechanism.

Detection – Canaries

- Detect attempts at overwriting the return address.
- Place a check value (‘canary’) in the memory location just below the return address.
- Before returning, check that the canary has not been changed.
- Stackguard: random canaries.
  - Alternatives: null canary, terminator canary
- Source code has to be recompiled to insert placing and checking of the canary.

Canaries


return address
canary
buffer for variable A
check value
write to A:
value1|
value2|
my_address
to A
my_address
value2 ≠ check value
value1
attack detected

Randomization - Motivations

- Buffer overflow and return-to-libc exploits need the virtual address to which pass control:
  - Address of the attack code in the buffer.
  - Address of a standard kernel library routine
- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine.
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine.

ASLR Defense

- Arranging the positions of the key data areas randomly in a process address space.
  - e.g., the base of the executable and position of libraries (libc), heap, stack
  - Effects: for return to libc, need to know address of the target function.
  - Attacks:
    - Repetitively guess randomized address (brute-forcing attack)
    - Spraying injected attack code
- Vista has this enabled, software packages available for Linux and other UNIX variants

PaX ASLR

- Address Space Layout Randomization (ASLR) renders exploits which depend on predetermined memory addresses useless by randomizing
- PaX implementation of ASLR consists of:
  - RANDUSTACK
  - RANDKSTACK
  - RANDMMAP
  - RANDEXEC

Prevention – Safer Functions

- C is infamous for its unsafe string handling functions: **`strcpy`**, **`sprintf`**, **`gets`**, …
- Example: **`strcpy`**

```
char *strcpy( char *strDest, const char *strSource );
```

  - Exception if source or destination buffer are null.
  - Undefined if strings are not null-terminated.
  - No check whether the destination buffer is large enough.

Prevention – Safer Functions

- Replace unsafe string functions by functions where the number of bytes/characters to be handled are specified:

  `strncpy`, **`_snprintf`**, **`fgets`**, …

- Example: **`strncpy`**

```
char *strncpy( char *strDest, const
char *strSource, size_t count );
```

- You still have to get the byte count right.
  - Easy if data structure used only within a function.
  - More difficult for shared data structures.

Prevention – Filtering

- Filtering inputs has been a recommended defence several times before in this course.
- Whitelisting: Specify legal inputs; accept legal inputs, block anything else.
  - Conservative, but if you forget about some specific legal inputs a legitimate action might be blocked.
- Blacklisting: Specify forbidden inputs; block forbidden inputs, accept anything else.
  - If you forget about some specific dangerous input, attacks may still get through.
- Taint analysis: Mark inputs from untrusted sources as tainted, stop execution if a security critical function gets tainted input; sanitizing functions produce clean output from tainted input (.i.e SQL injection).

Prevention – Filtering

- White lists work well when valid inputs can be characterized by clear rules, preferably expressed as regular expressions.
- Filtering rules could also refer to the type of an input; e.g., an `is_numeric()` check could be applied when an integer is expected as input.
- Dangerous characters can be sanitized by using safe encodings.
  - E.g., in HTML <, > and & should be encoded as `<`, `>`, and `&`.
- Escaping places a special symbol, often backslash, in front of the dangerous character.
  - E.g., escaping single quotes will turn d'Hondt into d\'Hondt.

Prevention – Type Safety

- Type safety guarantees absence of untrapped errors by static checks and by runtime checks.
- The precise meaning of type safety depends on the definition of error.
- Examples: Java, Ada, C#, Perl, Python, etc.
- Languages needn't be typed to be safe: LISP
- Type safety is difficult to prove completely.

Detection – Code Inspection

- Code inspection is tedious: we need automation.
- K. Ashcraft & D. Engler: Using Programmer-Written Compiler Extensions to Catch Security Holes, IEEE Symposium on Security &Privacy 2002.
- Meta-compilation for C source code; 'expert system' incorporating rules for known issues: untrustworthy sources ⌈ sanitizing checks ⌈ trust sinks; raises alarm if untrustworthy input gets to sink without proper checks.
- Code analysis to learn new design rules: Where is the sink that belongs to the check we see?
- Microsoft has internal code inspection tools.

Detection – Testing

- White box testing: tester has access to source code.
- Black-box testing when source code is not available.
- You do not need source code to observe how memory is used or to test how inputs are checked.
- Example: syntax testing of protocols based on formal interface specification, valid cases, anomalies.
- Applied to SNMP implementations: vulnerabilities in trap handling and request handling found http://www.cert.org/advisories/CA-2002-03.html
  - Found by Oulu University Secure Programming Group http://www.ee.oulu.fi/research/ouspg/

Mitigation – Least Privilege

- Limit privileges required to run code; if code running with few privileges is compromised, the damage is limited.
- Do not give users more access rights than necessary; do not activate options not needed.
- Example – debug option in Unix sendmail: when switched on at the destination, mail messages can contain commands that will be executed on the destination system.
- Useful for system managers but need not be switched on all the time; exploited by the Internet Worm of 1988.

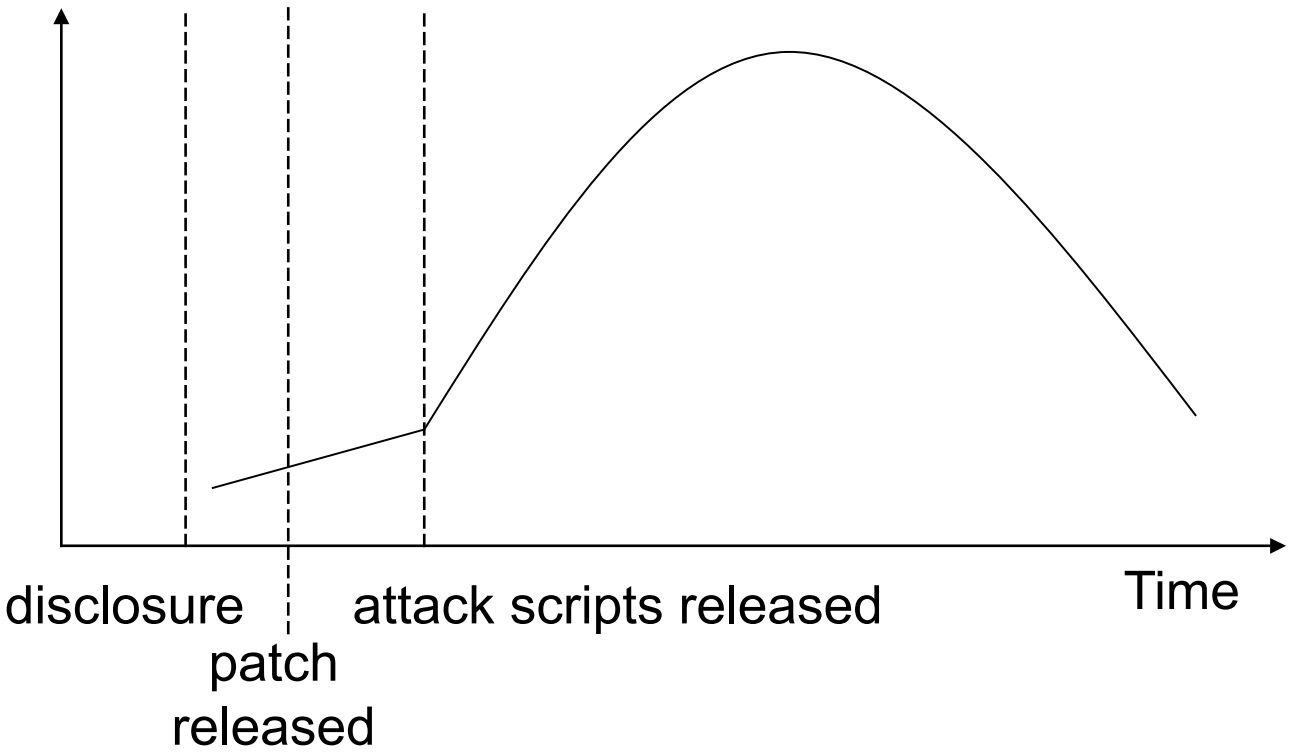
Lesson Learned

- In the past, software was shipped in open configurations (generous access permissions, all features activated); user had to harden the system by removing features and restricting access rights.
- Today, software often shipped in locked-down configurations; users have to activate the features they want to use.

Reaction – Keeping Up-to-date

- Information sources : CERT advisories, BugTraq at www.securityfocus.com, security bulletins from software vendors.
- Hacking tools use attack scripts that automatically search for and exploit known type of vulnerabilities.
- Analysis tools following the same ideas will cover most real attacks.
- Patching vulnerable systems is not easy: you have to get the patches to the users and avoid introducing new vulnerabilities through the patches.
- Is the software patching real secure?

Intrusion Patterns



W. Arbaugh, B. Fithen, J. McHugh: Windows of Vulnerability: A Case Study Analysis, IEEE Computer, 12/2000

Broken Abstractions

- Treating the problems presented individually, would amount to penetrate-and-patch at a meta-level.
- We looking for general patterns in insecure software, we see that familiar programming abstractions like variable, array, integer, data & code, address, or atomic transaction are being implemented in a way that breaks the abstraction.
- Software security problems can be addressed
  - in the processor architecture,
  - in the programming language we are using,
  - in the coding discipline we adhere to,
  - through checks added at compile time (e.g. canaries),
  - and during software development and deployment.

Summary

- Many of the problems listed may look trivial.
- There is no silver bullet:
  - Code-inspection: better at catching known problems, may raise false alarms.
  - Black-box testing: better at catching known problems.
  - Type safety: guarantees from an abstract (partial) model need not carry over to the real system.
- Experience in high-level programming languages may be a disadvantage when writing low level network routines.