Source Code Vulnerabilities

Angelos Keromytis
Columbia University
Code vulnerabilities

- Protocols and algorithms may be perfect
  - Implementations is another story!
- Majority of vulnerabilities are result of bad code
  - Buffer overflows
  - Race conditions
  - Insufficient/wrong argument validation
  - SQL injection
- Backdoors, trojan horses
Applicability

- Applications
  - Usually privileged ones
- Extensible (operating) systems
- Mobile agents

- Malicious code, viruses
Buffer overflows

- Overwrite return pointer in caller’s stack frame
  - Arguments on the stack
  - Missing bounds checking
- BSS and heap overflows
  - Virtual functions, object methods
- Jump-into-libc
- The goal is to transfer the control flow to injected code
  - Or to existing code, with arguments of attacker’s choice
int main(int argc, char **argv) {
    char fname[] = "/tmp/testfile";
    char buffer[16];
    u_long distance;

    distance = (u_long)fname - (u_long)buffer;
    printf("fname = %p
buffer = %p
    distance = 0x%x bytes\n",
           fname, buffer, distance);
    printf("fname = %s\n", fname);
    strcpy(buffer, argv[1]);
    printf(fname = %s\n", fname);
    return 0;
}
Example heap overflow

```c
int main() {
    u_long distance;
    char *buf1= (char *)malloc(16);
    char *buf2= (char *)malloc(16);

    distance= (u_long)buf2 - (u_long)buf1;
    printf("buf1 = %p\nbuf2 = %p\ndistance = 0x%x bytes\n", buf1, buf2, distance);
    memset(buf2, 'A', 15); buf2[15]='\0';
    printf("buf2 = %s\n", buf2);
    memset(buf1, 'B', (8+distance));
    printf("buf2 = %s\n", buf2);
    return 0;
}
```
Example SQL injection

- Dynamically generated queries

```
"select * from mysql.user
 where username=' " . $uid . " ' and
 password=password(' " . $pwd . " ');
```

- Feed bad input

```
"select * from mysql.user
 where username='" or 1=1; -" and
 password=password('_any_text_');"
```
Race conditions

• Time Of Check To Time Of Use (TOCTTOU) bugs
• Example of updating /etc/passwd
  • Pick "random" filename
  • Check that it does not exist in /tmp
    • If it does, loop
  • If not, open file
  • Copy contents of /etc/passwd
  • Add new entry
  • Copy temp file to /etc/passwd
• Other example: changing symbolic link pointer between check and use
Bad argument validation

• Example: sendmail debug flag
  • Given as number in command line
  • Used as index in table to set appropriate debug flag
  • But: no bounds checking
  • And: sendmail running "setuid"
• Result: able to add code (and execute it)
• Example: sprintf format string
Parameters of proposed solutions

- Performance
- Coverage
  - Resistance to new attacks
- Ease-of-use
  - Intrusiveness in programming style
Code signing

- Code producer (or trusted compiler) digitally signs code
- User checks signature, verifies code comes from "trusted" entity

- In general, insufficient:
  - Implies "binary" trust model
  - Malevolent/subverted "trusted" party can cause damage
  - Lack of a PKI -> non-scalable approach

- Reasonable as first line of defense
Unix chroot()

- In unix, (almost) everything is part of the filesystem
- Limit what code/process can do by restricting their view of the filesystem
- Typically, daemon processes ran in their own mini-filesystem
- Possible to escape, or cause damage even from inside a chroot’ed environment
- FreeBSD jail()
  - Different virtual machine based on IP address
Capabilities

- Introduce fine-grained access control for all resources
- Allow users to specify exactly what resources processes have access to
  - Increased administrative complexity
  - Must modify existing applications
System Call Monitoring

• Sandbox untrusted applications by monitoring system calls
  • Enforce particular policy
• Policy may be uploaded to kernel
• Similar to virus checker
• Have to hand-tune policy for individual applications
  • Fine for widely-used daemons, tricky for downloaded code (e.g., plug-ins)
• Java security manager approach fundamentally similar
Static analysis

- Look at piece of code, determine faults
  - Manual inspection
  - Model checkers
- Inherently difficult problem
Dynamic analysis

- Augment static buffers with size information
- Propagate throughout program calls
  - Inject checks prior to use
- Very invasive, difficult to get right

- Different approach: Perl Taint model
Software Fault Isolation (SFI)

- Software encapsulation of code
- Partition code into data and code segments
  - Prevent self-modifying code
- Code is inserted before each load, store, and jump instruction
  - Verify that the target address is safe
- Done at compiler, link, or run time
  - Increases program size, slow down
- "Tricky" for CISC architectures
Compiler tricks

• First approach: instrument all pointer accesses
  • Expensive!
• StackGuard: inject runtime checks for buffer overflows
  • Use "canaries" to detect overflows
• StackShield: save return address to write-protected memory
  • Restore before return
• StackGhost: use processor (SPARC) register windows
Compiler tricks (cont.)

- **ProPolice**: similar to StackGuard, re-orders variables
- **FormatGuard**: wrappers for printf function family
- **Binary Rewrite**: redundant copy of return address
  - Inject checks directly into legacy programs
- **Not fool-proof**
  - Heap-based overflows, SQL-injection
- **Performance penalty** (sometimes significant)
Better APIs

- Engineering solution
  - `strcpy/strcat` -> `strncpy/strncat`
  - `sprintf` -> `snprintf`
  - `tmpnam` -> `mkstemp`
- Not always possible (thanks to standards)
  - Sometimes, new API confusing
    - `strlcpy/strlcat`
Better APIs (cont.)

- Libsafe: substitute suspicious functions with "safe" instances
  - sprintf, fgets, strcpy, strcat
- Does not catch other types of faults
Proof-carrying code

• Input: piece of code, safety policy
• Output: safety proof
• Proof generation is computationally expensive
  • Verification simpler and less expensive
• Compiler need not be trusted
  • Only the verifier
Proof-carrying code (2)

- Burden is on the code producer
  - Prove once, use everywhere (with same policy)
- Reliance only on the verifier (which is small)
- Tamperproof programs: modifying a program will
  - Invalidate the proof
  - Make the proof non-applicable to the program
  - Proof and program still valid -> good
- Simple programs (packet filters) / policies
  - Promising
Safe languages

- Use a language where "bad thoughts" are impossible
- Examples: Java, ML/Caml, Erlang, etc.
  - Type safety
  - Memory management
- VM may still be unsafe (Java bytecode, JIT, ...)
- User reluctance to learn a new language
- "Too different from C"
  - Cyclone
- CCured
  - Static analysis + runtime inspection
Code Randomization Techniques

- Apply Kerckhoff’s principle on programs
  - Key-driven randomization of certain aspects of binary
  - Reveal key to OS
  - Attacker must mount exhaustive-search attack

- Randomize location/size of stack/activation records
- Randomize location of linked libraries
- Randomize instruction set!