16. Automatic Exploit Generation & Obfuscation

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

http://ter.ps/757
https://www.facebook.com/SDSAtUMD

Today’s Lecture

• Where we’ve been
  – Authentication and access control
  – Internet security
  – Malware distribution networks
  – SSL/TLS and the PKI

• Where we’re going today
  – Starting last module: Distributed Infrastructures Supporting Cybercrime
  – Defenses against vulnerability exploits
  – Advanced exploitation techniques

• Where we’re going next
  – Worms
Expectations Survey Results

- Project concerns
  - Projects expected to go in more depth in their topic than covered in class
  - Come talk to me about the projects

- Workload concerns
  - 31 papers, max 3 papers / week
  - Toward low end for a systems-oriented graduate course
  - Learn extract the important points from each paper quickly
    - Important skill for grad school

- Topic requests
  - Will note in future lectures

- Format suggestions
  - Reading due date

Recall: Buffer Overflow Exploits

- Hijack the program control
  - How?

- Ensure that the attack code is stored somewhere in memory
  - How?
Fundamental Causes for Basic Stack Smashing Exploits

- C strings are nul-terminated, rather than specifying the bound
  - Programmer must check the range manually
  - Many unsafe functions in the standard C library
    - strcpy(char *dest, const char *src)
    - strcat(char *dest, const char *src)
    - gets(char *s)
    - scanf(const char *format, ...)
    - printf(const char *format, ...)

- Stacks grow down and arrays grow up

- Von Neumann architecture: program and data in same memory
  - In addition, for x86: no distinction between executable and readable pages

Defense #1: StackGuard
[Cowan et al., USENIX Security 1998]

- Goal: Prevent attacker from overwriting return addresses
- Embed canaries (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

  \[\text{buf} \quad \text{canary} \quad \text{sfp} \quad \text{ret addr}\]

  Local variables
  Pointer to previous frame
  Return execution to this address

- Choose random canary string on program start
  - Attacker can’t guess what the value of canary will be
  - Example: /GS option in the .NET compiler
- Terminator canary: “\0”, newline, linefeed, EOF
  - String functions like strcpy won’t copy beyond “\0” => exploit won’t leave the canary as “\0”
**StackGuard Implementation**

- StackGuard requires code recompilation

- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server

- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient

---

**Defeating StackGuard**

- Suppose program contains `strcpy(dst,buf)` where attacker controls both `dst` and `buf`
  - Example: `dst` is a local pointer variable
- Can overwrite other function pointers
  - Exception handlers, virtual methods in C++
Defense #2: Address Space Layout Randomization (ASLR)

- **Goal:** Prevent attacker from guessing where stack frames and executable functions are in memory (**why needed?**)

- Randomize the **starting address** of the stack and heap and the location where shared libraries are mapped

- Other randomization methods
  - Randomize system call ids or instruction set

Recall: Return-to-libc Attack

- Jump to a function in libc
  ```c
  int system(const char *command) {
  – system() invokes a UNIX command (e.g. /bin/sh)
  – You can put the command on the stack
  }
  ```

- Limitations
  - 0 bytes to terminate command strings
  - Some functions take args. from registers
    - Modifying the registers requires you to inject code
ASLR Deployment

• Booting Vista twice loads libraries into different locations:

<table>
<thead>
<tr>
<th>Library</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printdln.dll</td>
<td>0x69D7F0000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntmap.dll</td>
<td>0x7537F0000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrd.dll</td>
<td>0x6F20C0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x76160000</td>
<td>Microsoft OLE for Windows</td>
</tr>
<tr>
<td>printdln.dll</td>
<td>0x69DA90000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntmap.dll</td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrd.dll</td>
<td>0x6D9D0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x763C0000</td>
<td>Microsoft OLE for Windows</td>
</tr>
</tbody>
</table>

• Deployment
  – Windows Vista: 8 bits of randomness for DLLs
    • Aligned to 64K page in a 16MB region => 256 choices
  – Linux (via PaX): 16 bits of randomness for libraries
  – More effective on 64-bit architectures

Defeating ASLR

• On Vista: 1 out of 256 guesses will succeed
  – Targeted attack: May be able to send multiple inputs to remote program
    • Program may restart automatically after crash
  – Large scale attack: Hacking 1/256 of the vulnerable hosts is not bad

• ASLR is only applied to images for which the dynamic-relocation flag is set
  – ASLR requires DLLs that are memory independent

• Information disclosure bug (buffer overflow disclosing the value of a pointer) allows attackers to figure out the random offset

• ASLR prevents return-to-libc attacks
  – Can still inject code on stack and execute it
Defense #3: Code ≠ Data

- **Goal**: prevent execution of injected code

- Several ways of preventing the execution of **untrusted code**
  - Make stack and other data areas non-executable
    - Unfortunately, this messes up useful functionality (e.g., Flash, JavaScript)
      - **Why**?
    - Digitally sign all code
    - Ensure that all control transfers are into a trusted, approved code image

W⊕X / DEP

- **Mark all writeable memory locations as non-executable**
  - Example: Microsoft’s Data Execution Prevention (DEP)
  - This blocks (almost) all code injection exploits

- **Hardware support**
  - AMD “NX” bit, Intel “XD” bit (in post-2004 CPUs)
  - Makes memory page non-executable

- **Widely deployed**
  - Windows (since XP SP2),
  - Linux (via PaX patches),
  - OS X (since 10.5)
What Does DEP Not Prevent?

- Can still corrupt stack ...
  - ... or function pointers or critical data on the heap

- As long as the saved return address points into existing code
  DEP will not block control transfer

- ASLR and DEP must be used together
  - ASLR prevents jumps to library functions
  - DEP prevents executing code on the stack

- Used together, ASLR and DEP prevent many attacks
  - Attacker cannot execute arbitrary code, especially if system() is not available
  - Is this the end of exploits?

Return-to-libc on Steroids

- Overwritten saved EIP need not point to the beginning of a library routine

- Any existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction

- What if instruction sequence contains RET?
  - Read the word pointed to by stack pointer (ESP)
  - Use it as the new value for EIP
  - Increment ESP to point to the next word on the stack

- Attacker loads the stack with addresses that point to instruction sequences ending in RET
Chaining RETs for Fun and Profit

- Can chain together sequences ending in RET [Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique”, 2005]
- What is this good for?
  - Answer [Shacham, “The Geometry of Innocent Flesh on the Bone”, CCS’07]: everything
    - Turing-complete language
    - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
    - Attack can perform arbitrary computation using no injected code at all (bypass DEP)
      - Or change NX state of memory page
    - Use only RETs without CALLs:
      return-oriented programming (ROP)

Defense #4: Control Flow Integrity (CFI) [Abadi, Budiu, Erlingsson, Ligatti, CCS’05]

- Goal: prevent arbitrary control flow transfers
- In most software, indirect control transfers (e.g. function pointers, returns) have only a few valid targets
  - Determine the control flow graph (CFG) ahead of time
    - Source code analysis
    - Binary analysis
    - Execution profiling
  - Instrument sources and destinations of computed control flow transfers
- Method
  - Insert a CFI label (distinct bit pattern) at each destination
  - Before source, insert dynamic check that destination has the right CFI label
CFI Security

• CFI is effective against attacks based on illegitimate control-flow transfer
  – Smashing the return address
  – Smashing function pointers
  – Return-to-libc
  – Return oriented programming
  – Sandbox escape
  • Recall: SFI inserts dynamic checks to ensure that target memory accesses are constrained to a certain range
  • CFI makes these checks non-circumventable

• CFI is not widely adopted
  – Overhead is a concern

Other Targets of Memory Exploits

• Configuration parameters
  – Example: directory names that confine remotely invoked programs to a portion of the file system

• Pointers to names of system programs
  – Example: replace the name of a harmless script with an interactive shell
  – This is not the same as return-to-libc

• Branch conditions in input validation code

• None of these exploits violate the integrity of the program’s control flow
  – Only original program code is executed!
Example: Web Server Security

- CGI scripts are executables on Web server that can be executed by remote user via a special URL
  - http://www.server.com/cgi-bin/SomeProgram

- Don’t want remote users executing arbitrary programs with the Web server’s privileges, need to restrict which programs can be executed

- **CGI-BIN** is the directory name which is always prepended to the name of the CGI script
  - If CGI-BIN is "/usr/local/httpd/cgi-bin”, the above URL will execute /usr/local/httpd/cgi-bin/SomeProgram

Other Exploitation Techniques

- API misuse (common problem with crypto libraries)
  - Common problem with crypto libraries
  - Example: incorrect SSL certificate validation

- These exploits do not involve memory corruption
Example: SSL Certificate Validation

Hello

I am PayPal.com (or whoever you want me to be)

Here is PayPal’s certificate for its RSA signing key
And here is my signed Diffie-Hellman value

Validate the certificate

... then verify the signature on the DH value using the public key from the certificate (why?)

“Goto Fail” Bug in Apple SSL Stack

Here is PayPal’s certificate
And here is my signed Diffie-Hellman value

... verify the signature on the DH value using the public key from the certificate

if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
goto fail; ??
err = sslRawVerify(...); [Signature is verified here]
... fail: ... return err ...
Review of Lecture

- What did we learn?
  - Defenses against memory corruption: stack canaries, DEP, ASLR, CFI
  - Limitations of these defenses

- Sources
  - Vitaly Shmatikov, Dan Boneh, Lujo Bauer

- What’s next?
  - Worms and infection spreading