8/29 Buffer Overruns

Max Feldman

August 8, 2012

Buffer overrun errors used to be the #1 most common vulnerability (may be replaced by web apps)

Intro discussion

- Sysadmins often run services as root, so vulnerabilities in those are a serious source of problems
- Not just server code - browser, email clients, most PCs are single user now, so even non-root programs are a concern
- Music players, java, many other programs are a concern
- 3 major sources of vulnerabilities
    - vulnerable code
    - misconfiguration, out-of-date software, etc
    - "human element"- phishing, others
- malware threats include drive-by-downloads and social engineering
    - disagreement on prevalence of each

Defenses against buffer overrun

- runtime tools
- bounds checking, don't write past length of buffer
    - has overhead - depends (apache- little overhead, I/O bound, much more overhead for CPU bound programs)
- baggy bound checking: 1/16 of addr space is reserved for shadow data structure
- page faults not a significant contributor to baggy bounds overhead
- 100-1000 cycles to go to main memory due to cache miss
- Randomize syscall numbers, so attackers can’t guess as easily
- developer education so vulnerabilities are not introduced (although legacy code is still an issue), use more secure methods
- Static analysis to detect bugs- no performance overhead at runtime
- Could go further, prove no buffer overruns with formal methods
    - much harder without the source code
    - may not find all bugs
- safer/newer language (eg. Java)
    - may not be as fast as C
    - but java's type checking can eliminate some bounds checking needs

Defenses and Security Evaluation

1. **Stack Canary**
    - placed between buffer and return address on stack
    - only helps with stack smashing
    - if attacker can learn canary's value then this is not useful (possible to learn this value through format string vulnerability, bug chaining)
    - good defense at the time it was proposed
    - Has been deployed

2. **Non-executable stack**
    - Does not provide defense for arc injection
    - low overhead
    - some applications may execute off the stack, but it is possible to handle such cases
    - Still possible to store malicious code somewhere other than the stack

3. **DEP (W^X)**
    - every page in memory is either writable or executable
    - arc injection still possible
    - OS loader- marks data as nonexecutable
    - Has been deployed

4. **Shadow Stack**
    - 2 stacks instead of 1
    - one stack frame with program variables
    - one stack frame with return addresses
    - breaks compatibility- requires compiler modification
    - critical program state stored in stack can still get overwritten
    - local variable could be a function pointer which can still be overwritten
    - doesn't help the heap

5. **Valgrind memcheck**
    - Use valgrind to check for out-of-bounds memory accesses, or other memory errors
    - http://valgrind.org/

6. **DieHard**
    - Tool for providing “Probabilistic Memory Safety for Unsafe Languages:
    - http://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1086&context=cs_faculty_pubs

7. **ASLR**
    - Address space layout randomization (base addresses stored in random spaces)
    - can reveal address space with bug chaining
    - address space isn't that large- 32 bits, but many alignment requirements (page alignments, other restrictions), windows has ~8-10 bits of entropy for one of the code regions
        - it is therefore feasible to try many times (brute force) and still exploit
    - 64 bit architecture- more entropy, so it is harder to guess
    - Deployed

8. **Baggy Bounds Checking**
    - limitations: overhead, overruns overwriting function pointers within structs (object-granularity)
    - not deployed- tremendous implementation cost

9. **Address Sanitizer**
    - Tool for detecting memory errors
    - http://code.google.com/p/address-sanitizer/

Future discussion

- DEP + ASLR = effective at stopping many attacks, low performance overhead, but can break JIT (writes executable code to a data space)
- return-oriented-computing = generalization of return-into-libc, defeats DEP
- JIT spraying= defeats ASLR