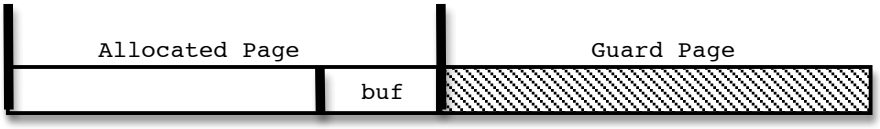
CS261 – Computer Security
Lecture 4
September 9, 2009
Scribe: Kevin Klues

Buffer Overrun Defenses:

1) Stackguard (MSVC# / GS)
   - At the time of its development, stack smashing was the #1 method of attack used by hackers to infiltrate security systems
   - Stackguard works by changing the stack frame organization to include a "canary" value just before the stack frame pointer on the current stack frame.

   - This canary value is checked to make sure it hasn't changed between the time a function is entered and the time that function exits.
   - If it's the same, then we know that no buffer overrun occurred and we can continue.
   - If its different, then we assume someone has done something malicious and we act accordingly.
   - Methods to overcome this security measure:
     - Write directly to the return address bypassing the canary value
     - Learn the canary value somehow and overwrite it with the correct value as you execute the buffer overrun attack
2) Non Executable Stacks
   - Use hardware protection mechanisms to ensure that the memory region where the stack resides is set to non-executable
   - Methods to overcome this security measure:
     - Arc-Injection attack (return from libc attack)
3) W^X (Write XOR Execute) (a.k.a. DEP or NX)
   - Maintain the invariant that all pages in memory are marked either writable of executable, but not both (as well as readable as appropriate)
   - One problem with this approach is that it rules out the use of dynamically generated code
     - Some systems have used op-outs to overcome this problem
     - The problem is that op-outs are used to liberally (i.e. IE6 and the JVM in Windows)
   - Methods used to overcome this security measure:
     - Arc-Injection attacks
4) Separate Stacks
   - Dedicate one for the function arguments
   - Another for the return addresses
   - Primary problem is the cost of maintenance that the fact that this approach doesn't help solve problems with heaap allocated buffers

5) Electric Fence
   - Place a guard page at the end of buffers allocated on the heap.



   - Naïve approach aligns each allocated buffer on the left side of an allocated buffer, wasting all space to the right of each buffer up to the end of a page
   - Other approach is to align the buffer to the right, placing the guard page just after the end of the allocated buffer.
   - Major downside of this approach:
     - Performance and fragmentation overhead
   - Methods used to overcome this security measure:
     - Only works, for the heap, so says nothing about stack allocated buffers
     - Function pointers placed just after a buffer inside a struct can still be overwritten without warning
     - Approach doesn't help with format string attacks
     - Could potentially write backwards if the code allowed it (but this could easily be solved by placing a guard page both before and after the allocated buffer)
6) Valgrind Memcheck
   - Works on binaries rather than source code
   - Works like a simulator in which your executable executes
   - Tracks allocated regions in a virtual address space
   - Conceptually maintains T[a] – a bitmap of of every byte address in the address space (takes up space to maintain, however – 2^29 bits worth!)
     - Bit set to 1 if its been allocated by malloc
     - Set to 0 otherwise
   - Every access to memory check T[a] to make sure you are actually accessing a byte in memory that has previously been allocated using malloc
   - Methods used to overcome this security measure:
     - Only works, for the heap, so says nothing about stack allocated buffers
     - What if its possible for us to change the T[a] array itself?
       - Fix by unmappoing the first 2^26 bytes in the virtual address space, causing a fault to occur if any of these bytes are written
       - Not 100% sure on the details of this (check the internet for more information)
     - Only done at the granularity of malloc, so doesn't protect full objects
   - If combined with other approaches, this method could be extremely effective
   - Performance and memory overheads are overwhelming though
7) ASLR
   - Randomize the start of each segment so that attacks must be customizes on a program by program basis

- Methods used to overcome this security measure:
    - Brute force could be used if the only harm done is that the program restarts if an improper address is written to (could also be used to exploit a DoS attack
    - Since random offsets are always a multiple of a page size, there are less bits than expected with which to randomize placement
        - Vista – 8 bits for all dlls
        - 14 bits for stack allocations
        - 5 bits for heap allocations
    - If we can exploit buffer overruns to read past the end of an array and read the frame pointer, then we could learn enough info to infer a particular random offset at runtime
    - Trick the program to malloc lots of memory, taking up most of the remaining space in the virtual address space
        - Since we would then control whats in each buffer, we could place noops at the beginning of these buffers (noop sled), followed by real code placed near the end of buffers hoping we are placed near libarary code located somewhere in the address space
        - Reduces the probability of missing
        - Easy to do with Javascript in web browsers
- Question: Why isn't malloc implemented to randomize allocations on the heap, or is it?
    - This could increase security
    - Possible course project to look into this

8) Baggy Bounds Checking
    - How could we attack a system that had baggy bound checking turned on?
        - Uninstrumented code linked against code compiled with baggy bounds checking is still vulnerable
        - This method doesn't overcome the problems where structs contain buffers followed by function pointers and overflows overwrite their value
    - Class discussion on this paper:
        - Weaknesses of fragmentation not discussed very exhaustively in the paper
            - Impossible to have exernal fragmentation, however, since a buddy allocation scheme is used
            - Internal fragmentation might be a BIG problem, however
        - Is this method used in productaion systems, or only for research and debugging?
            - Not too much, because there is a large performance overhead in using it
            - That said, this technique is MUCH better than previous approaches
        - Only really necessary because its hard to do bounds checking in C because of the existence of pointers

- Problem with previous approaches:
    - Fat pointers: pointers passed around together with their bound annotations break the ABI and cannot work with existing code
    - Splay Tree techniques stored in a shadow data structure were too slow and had a very large overhead in terms of memory usage
- Orthogonal approach is the use of explicit programmer annotations to place bounds on all pointers declared in a program (e.g. IVY)

9) Taint Tracking
    - Primary contribution of this paper was to provide taint tracking mechanisms at the byte granularity rather than at the "input size" granularity
    - Why is this important?
        - Allows us to distinguish exactly which portion of a 'tainted" query is the tainted part, rather than being required to mark the entire query or assignment as tainted
    - The approach taken here is pretty much ideal, but previous systems were always too slow to be usable
    - Policies mentioned in the paper that we think we could break:
        - Format String attacks: (%[a-z A-Z])+
            - Break with %5n
            - False positive with %%x
        - Tainted SQL Query (OR)+ | (SELECT)+ | (AND)+ | …..
            - Blacklisting like this can be really hard to get right
            - White lists are much better
            - Fix: Parse the SQL query first and make sure that the tainted data is limited to parts interpreted as data, not control commands