Defending Against Derandomization Attacks

Patrick Graydon
Motivation: Meet Jane Whitehat

• Jane’s servers provide a critical service
  – If service is interrupted for more than a few minutes at a time, Bad Things happen:
    • Business may grind to a halt
    • Fortunes may be lost
    • Careers may be ruined…
    • Lives may be in jeopardy?
Jane’s services

• Jane’s servers are a juicy target
  – Maybe Jane’s employers have enemies…
  – Maybe the data on Jane’s servers is very valuable…
  – Maybe Jane’s servers are just a way to break into the company’s network and go for the big score
Know thy enemy

- Meet John Blackhat, l337 h4x0r
- John wants into Jane’s network
  - He has motivation
  - He knows his trade:
    - Stack smashing attacks
    - Return to libc attacks
    - ...
  - He has corporate/national backers???
Jane’s defense

• Jane protects her services by deploying instruction set randomization
  – Or maybe StackGuard…
  – Or address space randomization…
  – Or maybe all of the above…
Randomization defenses

• Any of these techniques work by randomizing some part of the running system
  – Attacker must guess a key before an attack succeeds
  – In ISR on Jane’s Intel Heptium-based servers, this is a 16-bit value XORed with the instruction stream
Keys can be guessed at

• Remember *War Games*? After enough guesses you too could start WWIIII…

CPE1704TKS

• Conclusion:
  Jane Needs Our Help!
Sometimes failure is good

- Randomization defenses share some characteristics that may help us help Jane:
  - Any given attack attempt is likely to fail
  - Any failed attempt is likely to crash the system

- Why is this good?
  - Because it gives us a way to identify attack attempts (After the fact, though)
Can we find the attacker?

• Thought: maybe we can identify the attacker’s IP address and block it
  – Easy to implement; minimal impact on users
  – What about spoofing?
    • If Jane’s services require a full TCP connect, it is hard (but not impossible) to spoof the address…
  – But there are other problems…
Problems with blacklisting…

• What if John has written a worm?
  – Attacks could be coming in from $> 2^{14}$ compromised hosts at the same time…

• What if John employs a zombie network?
  – Ditto

• What if the attacker has access to the intranetwork wiring?
  – Could pretend to be a legitimate client…
Maybe we can duck and cover?

- In some we can detect an attack and suspend service until the sysops can react...
  - Could take an eternity ... maybe even minutes 😊
  - Bottom line: Jane doesn’t have the luxury of being able to do this
When all else fails...

• Maybe we can learn something about the requests themselves
  – If we could find a pattern in the requests that cause crashes, we could block attacks as they happen
Assumptions I’ve made

• Jane’s service receives requests as binary blocks over the network and sends one reply to each.
• Jane’s service is written in C/C++ and has a buffer overflow hole (oops!)
• John has access to the code for the software Jane uses.
More assumptions…

• Jane is willing to make modifications to the server software at the source level
  – Maybe we could do this at the OS level, but we’ll cross that bridge when we get there…

• Jane’s servers are configured to reboot the server software when it crashes
Knowing when to learn…

- We modify Jane’s software so that:
  - Before accepting a request, it checks against known attack signatures and saves the incoming request as a ‘suspicous’ request file
  - After servicing a request, it marks the suspicious request as good instead
  - When it starts up, it looks for a suspicious request file—if it’s there, the process must have crashed, and the file contains the request that crashed it
Finding the patterns

• We can look for areas of a request that match other known attack requests
  – First, sort a pair of requests by byte values
  – Look at each matching pair of bytes in turn:
    • Are they part of a matching regions in the unsorted request streams? If so, take as big a region as we can get...
  – Basically, this is what grep does...
Finding too much

- It is possible to find too big a pattern
  - We want patterns that are general enough to catch all similar attacks but specific enough not to generate false positives...
  - We can solve this by finding patterns in the patterns
Doesn’t ISR pose a problem?

- Not really, you just need to do two scans for signatures:
  - The first is the normal scan, and it picks up unencrypted things like static strings and the target buffer address
  - The second is on a special de-ISR’d version of the requests:
Yeah, yeah, but does it work?

• Well, sort of…
  – Tested using:
    • 4,388 randomly generated 64-to-512 byte ‘good’ requests (512 byte buffer, did 1,000 at startup),
    • 275 simple buffer-overflow attacks straight out of *Smashing the Stack for Fun and Profit* tweaked for 16-bit XOR ISR, and
    • 337 attacks of my own devious design
    • A minimum of 6-byte keys
And???

• The good news:
  – The algorithm detected all but 4 of the stack smashing attacks and blocked them, and
  – Not one false positive!

• The bad news:
  – My devious test was a bit too devious—the algorithm failed to block a single instance
How devious is devious?

• The devious requests need only 6 non-random bytes:
  – Two bytes of code: EB FE
  – Four bytes of data: the location of the injected code, partially randomizable
  – The rest of the attack buffer can be completely randomized—no patterns to find!
  – Note: this only tells you what the randomization key is, it doesn’t get you in
Can we block the devious case?

• Probably not with this algorithm
  – Tried reducing the minimum key size to 4 bytes
  – Algorithm blocked 150 of 347 attack attempts (that’s 43% and it’s still better than nothing)
  – BUT at the cost of a false positive
    • 0.02% false positives, but may still be too much
Any performance impact?

• Some:
  – Loading the data sets took ~15 ms and didn’t seem to scale badly (and that’s good)
  – Requests examined in no time at all (well, in no time that could be measured on my Wintel box…)
  – What about signature identification speed?
Signature identification speed

![Graph showing the relationship between time (s) and the number of 'bad' samples. The x-axis represents the number of 'bad' samples ranging from 0 to 250, and the y-axis represents time ranging from 0 to 2.5 seconds. The data points suggest a trend where time increases as the number of 'bad' samples increases.]
So what now?

• Well, the signature detection algorithm isn’t perfect, but it may still be useful
  – Need to try this on other kinds of attacks
    • Should try a return-to-libc on a Fedora box…
  – Need to try this with real requests
    • Actual sever requests may be more or less similar to the attack patterns than my random generations
  – Need to experiment with techniques for managing the size of the good and bad sets
Any questions???