Robustness

Protocol Design

Network Protocols need to be robust…

- During normal operation
- Against simple failures
  - Packet being dropped (of course!)
  - Link going down
  - Node going down (and taking its memory = state with it)
- Against malfunctions
  - Hardware problems causing incorrect operation
    - Link errors occur often enough that we use checksums to reduce their probability
  - Implementation errors (bugs, ambiguities in the specification)
  - Heterogeneity
  - Configuration errors
- Against malice
  - Attacks intending to cause damage (“Denial of Service”)
  - Attacks intending to subvert access control

Robustness against simple failures

- Bit errors are converted into dropped packets
  - CRCs are pretty good checksum, if used correctly
- Dropped packets are retransmitted
  - Reliability at L4 (and possibly L2).
- Link and node failures are typically handled at L3
  - Job of the routing protocol to find an alternate path
- Applications on failed nodes don't need the network
  - "Fate sharing": Application and transport state go away at the same time
- Distributed applications need to recover from node failures
  - This is its own subject in CS: Distributed Systems

Robustness against malfunctions and malice

- Much harder!
- We can no longer think in "probabilities"
  - An "improbable" malfunction may become the "preferred" attack vector
- Generalized CS approach: "Byzantine Generals"
  - Theoretical results show that system breaks down if > 1/3 of nodes malfunction
  - Solutions are typically very heavyweight

Robustness During Normal Operation

Traffic Surges in Multiparty Environments

- Need not be multicast protocols
  - Meshes of point-to-point relationships do as well
  - As do individual relationships with the same peer (e.g., some server)
- Synchronization
  - Coupled systems tend to synchronize
  - Explicitly care for randomization/dithering
- Implosion
  - Positive or negative acknowledgements, state change notifications, …
- Transients
  - E.g. rebooting after failures

Case Study

- University of Wisconsin NTP server was hard-coded in ~700K appliances (routers, firewalls)
- Implementation bug: request is retransmitted after only 1s

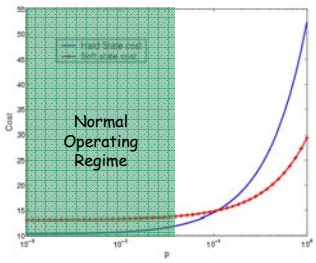
➔ Surge after non-reachability of server

  - ~500 Mbit/s request traffic

- Mitigation:
  - (a) Software update
    - Problem: is rarely deployed in private homes
  - (b) Replying to all requests to silence requester
    - Problem: reverse path may be fogged
  - (c) Remove hard-coded IP address from BGP routing system
    - Practical policy problem: can't eliminate a single IP address from routing tables
- Lesson learned: Want to have back-off mechanism!

Soft state:
Don't allow bad information to stick around

- Hard state approach: Agree on each state **change**, try to keep state in sync
  - Can always be made more efficient than soft state — if network conditions are known in advance
- Soft state approach: Keep desired state **alive**
  - State returns to **default value** after prolonged silence
  - "return to default value" message not even strictly required
  - Requires more base traffic
  - Less likelihood to go wrong in times of extreme stress



Design: Don't overoptimize

- Premature optimization is the root of all evil

  — Tony Hoare/Donald Knuth

- Optimization should be based on measurements
- (Of course, algorithms with exponential complexity etc. should be avoided from the outset, this is about micro-optimization…)

Robustness against Simple Failures

Beyond dealing with packet losses and the like

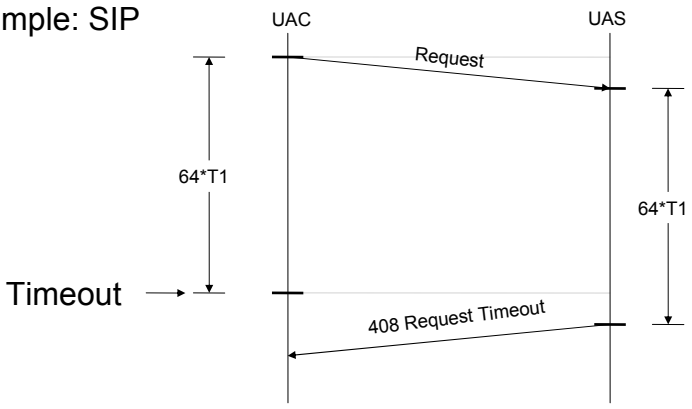
Timeouts

- Timeouts of the involved peers need to “match”
  - May be subject to misconfiguration
- Timeouts should be adaptive (see scalability)
  - Issue: independent measurements
- Timeouts may need to account for repeated packet loss
- Timeouts should be handled only on one side
  - Otherwise: if a timeout occurs, there is little point in saying so
- Issue: Slowness of application vs. problem to be handled
  - Application may be “swapped out”, computer may experience high load, …
  - Leads to delayed response
    - Can exacerbate server load

Notifying Timeout Not Necessarily Useful

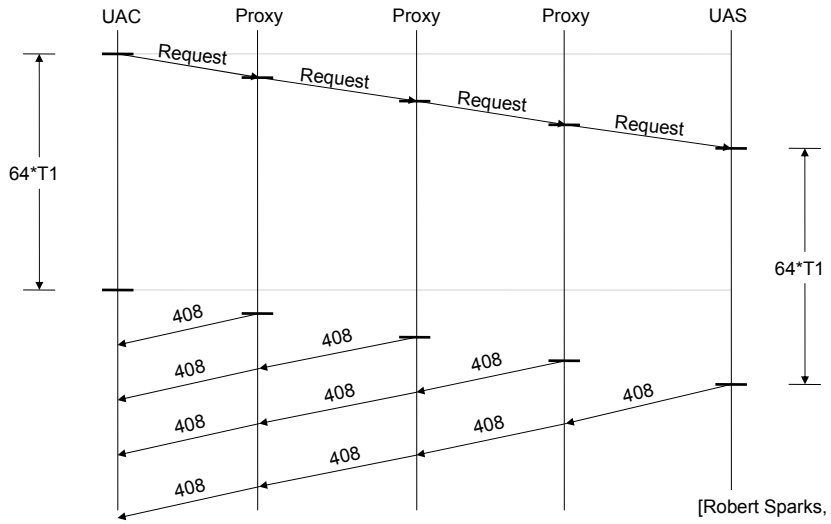
(If the requester has its own timer…)

Example: SIP



[Robert Sparks, 59th IETF]

The 408 Cascade “Storm”



[Robert Sparks, 59th IETF]

Fate Sharing

- If the application crashes, the network cannot help much
- If some random network element fails, the application should not need to care
- Couple / store application state only with the application
  - One aspect of the end-to-end principle

“The fate-sharing model suggests that it is acceptable to lose the state information associated with an entity if, at the same time, the entity itself is lost.”

[Dave Clark]

Case study: NFS (beyond fate sharing)

- NFS provides file service to a client
  - The files are persistent (by definition), the server state may be not!
- If client crashes: application went with it (fate sharing)
- If server crashes: application should be able to continue after server reboot

- NFS operates with a **stateless** server
  - All state is on the client
  - Handles handed out as intermediate result survive a server reboot
- Note: stateless ≠ connectionless
  - Modern NFS variants use TCP
  - Client can simply reconnect after a failure

Case Study: OSPF

- Broadcast networks:
  - Everyone would need to form an adjacency with everyone else
  - N × (N-1) / 2 adjacencies
    - Traffic
    - State
- Scalability:
  - Elect a Designated Router (DR)
  - Form adjacencies from everyone to DR only
- Robustness:
  - Also elect a Backup Designated Router (BDR)
  - Form adjacencies from everyone to BDR as well

Case Study: SIP

- Separates application from transport state
  - TCP connections may go up and down without harming dialog state
  - (Unlike: SMTP, POP3, IMAP4, FTP, Telnet)
- May separate dialog state (“in the network”) from media flow
- Allows for stateless operation of intermediaries
  - SIP stateless proxies
  - Forwarding decision is taken per request message
  - Response routing is done based upon **`Via:`** path in message

And the Hard Failures?

- Robustness requirements are application-specific
- Safety-critical applications
  - Banking, transportation, emergency responders, …
  - Stronger demand for synchronization, failover mechanisms, fail-safe properties, etc.
- Fault tolerance is a (CS) discipline of its own.

Robustness against Malfunctions

Case study: Arpanet 1980-10-27 (RFC 789)

- IMPs (routers) were 100 % busy handling routing updates
  - At a much higher rate than they "could" have been produced
- IMPs "could" only produce one routing update every 5 seconds
- Sequence number in 6-bit window made sure only the newest one would be sent on
- Hardware error created three copies with numbers 8, 40, 44
  - Each of these is "newer" than the previous one ➡ tight loop
- System was **not** self-stabilizing
  - Patched code had to be deployed to remove just the looping updates
  - Once system had stabilized, patch had to be removed again to resume normal operation

What can be learned from RFC 789?

- Systems should be **self-stabilizing**
  - Removing a malfunctioning system should return normal operation
  - Bad information, however it got into the system, should not survive indefinitely
- Assertions valid at one point in the system don't necessarily transfer to other points
  - If an IMP cannot generate more than one routing update every 5 seconds, this does not mean data of this kind cannot turn up in the network
  - Always consider the case that data might be bad: does the system stabilize?

Brittleness: Misconfiguration

- If two systems both have to be configured in a certain way to communicate successfully, **brittleness** ensues.
  - Misconfiguration is likely
  - Misconfiguration can lead to "half-working" states that are hard to detect
- Detect misconfiguration
  - Incompatible systems should refuse to appear to be talking
- Avoid misconfiguration
  - Remove options from protocols!
  - Use **negotiation** to agree on critical options

Negotiation: Ethernet

- Ethernet can be configured in dozens of variants
  - 10baseT, 100baseTX, 1000baseT
  - Half duplex, full duplex
  - Flow control options
- Different speeds just don't talk: good!
- Duplex mismatches **appear** to work until load becomes significant
- Ethernet has a negotiation protocol ("link pulses")
  - Can be switched off (another source of mismatches!)
  - Has been implemented in numerous incorrect ways

When failing hard is good

- A bad link may be worse than a dead link
  - Routing protocol may have a perfect alternative
  - But hello protocol may still make the link appear to work
- Links that turn bad look like they go down and up: "Flapping"
  - Can cause significant traffic (bits and CPU) in routing protocol
    - If 10 % of 200000 routes flap…
    - Overloading CPU of routers can cause cascading failures
  - BGP implementations have **route flap damping** to suppress flapping
    - Parameters hard to tune, though (bad settings made damping controversial)

Other Countermeasures:

- UDLD (Unidirectional Link Detection)
- LQM (Link Quality Monitoring, part of PPP)

Negotiation: TLS

- TLS ("SSL") peers need to negotiate crypto parameters
  - This needs to be done before full crypto is in effect
- Attack: interfere with negotiation
  - Mismatches result in interesting behavior
  - E.g., "negotiate-down" attack: convince both sides the other side has only limited crypto capabilities ("export version")
- Solution:
  - Agree on the exact result by exchanging signed statements about **all handshake messages** at the end of negotiation
- "Bidding down" attacks may occur in all kinds of security protocols

Negotiation: PPP

- PPP is probably the most configurable protocol
- Also very interoperable!
- Secret: LCP, NCPs negotiate all the options
  - Interoperable baseline (must implement)
- ConfigReq list **all** options desired
  - If not acceptable, peer can ConfigNAK, ConfigRej
  - Original proposer has to present another **complete** set in another ConfigReq
  - Peer echoes back the **complete** accepted set in an ConfigAck
- Occasional bugs in the negotiation convergence
- Very few bugs in misunderstanding of resulting configuration

Dealing with implementation differences

- IEN 111 (August 1979):

  The implementation of a protocol must be **robust**. Each implementation must expect to **interoperate** with others created by different individuals. While the goal of this specification is to be explicit about the protocol there is the possibility of differing **interpretations**. In general, an implementation *should* be **conservative in its sending behavior**, and **liberal in its receiving behavior**. That is, it should be careful to send well-formed datagrams, but *should* accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear).
- "should" became "must" in RFC 791 (September 1981)

Dealing with implementation differences

- RFC 1122 (October 1989), 1.2.2 Robustness Principle:

  At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability [RFC791]:

  "Be liberal in what you accept, and conservative in what you send"

  Software should be written to deal with **every conceivable error**, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue. In general, it is best to assume that the network is filled with **malevolent entities** that will send in packets designed to have the **worst** possible effect. This assumption will lead to suitable **protective design**, although the most serious problems in the Internet have been caused by unenvisaged mechanisms triggered by low-probability events; mere human malice would never have taken so devious a course!

Dealing with implementation differences

- Adaptability to change must be designed into all levels of Internet host software. As a simple example, consider a protocol specification that contains an enumeration of values for a particular header field -- e.g., a type field, a port number, or an error code; this enumeration must be assumed to be **incomplete**. Thus, if a protocol specification defines four possible error codes, the software must not break when a fifth code shows up. An undefined code might be logged (see below), but it must not cause a failure.
- The second part of the principle is almost as important: software on other hosts may contain deficiencies that make it **unwise to exploit legal but obscure** protocol features. It is **unwise to stray far from the obvious and simple**, lest untoward effects result elsewhere. A corollary of this is "watch out for misbehaving hosts"; host software should be prepared, not just to survive other misbehaving hosts, but also to cooperate to limit the amount of disruption such hosts can cause to the shared communication facility.

Dealing with implementation differences

- Jon Postel's "Robustness Principle": **be conservative in what you do, be liberal in what you accept from others**. [RFC793]
  - Paradoxical result: Tag Soup! [See also RFC3117]
- Formalisms such as XML Schemas can help pinpoint and thus minimize deviant behavior
  - Harder to do for behavior beyond syntax, though
- In the end, it's **interoperability**, not conformance, that counts
  - Early implementations leave an imprint that is best documented in an "implementer's guide"; can later go into draft standard
    - RTP ROHC, 168 pages, has implementer's guide with 24 pages (including 2 pages code)
  - RFC standards process is intended to weed out features where interoperability hasn't been demonstrated
- Interop events rather than compliance certification!

Robustness against Malice ("Security")

The four phases of an attack

- Reconnaissance
- Intrusion (using an "exploit")
  - May involve gaining initial access + escalation of privilege
- Consolidation, Cover up, Plant Backdoors (e.g., rootkit)
- Employment for objective

- Good security looks at all these phases:
  prevent, detect, contain

Spoofing

- Instead of subverting access control:
  Just pretend to be authorized
- Some systems only check source IP address
  - UDP: very easy to fake
  - TCP: more difficult, but in some cases still possible
- Session hijacking: take over connection **after** authentication
  - Simple countermeasures are part of TCP
    - Only protect against off-path attackers (subverted by eavesdropping)
  - Real protection requires cryptography

(D)DoS attack

- Denial-of-Service: Attacking the security objective availability
- Make a server crash
  - Use programming mistakes (e.g., unchecked buffers)
- Cause a system to go into circle-of-wagons mode
  - E.g., when accounts get closed after three wrong passwords
- Overload server (or network)
  - DDoS: Distributed DoS: Farm of “Zombies”, Botnet

Reflection/Amplification: Aiding in DDoS

- Reflection: Sending a “reply” to an unverified address
  - Can be used by attacker to hide identity
- Amplification: sending “back” **more**
  - Attacker needs less capacity to mount powerful DDoS attack
- Classic example: Smurf
  - Directed broadcast
  - Source address = victim
  - All destination hosts send “back” ICMP “port not reachable”

SYN-flood attack

- Objective: clog server
- Bonus: do this without giving hints about identity of attacker
- TCP: protected by three-way-handshake
  - Connection only is completed when peer answers with correct sequence number
  - Cannot easily fake source address
- Idea: just send SYN packet only
  - Easy to fake source address
  - Server needs to establish state (Timeout after several minutes)
- 1 Gbit/s ≈ 3×10^6 packets/s ≈ 0.5×10^9 half-open connections

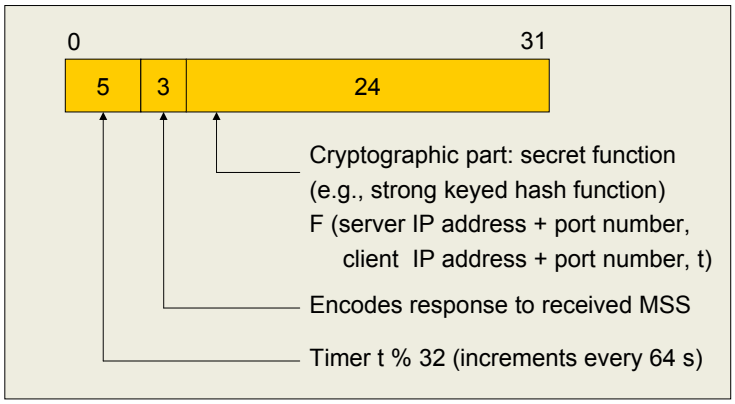
Countering Resource Depletion

- Attackers attempt to bind more resources on the target system than required to mount the attack
  - Make your system perform many and/or expensive computational operations
    - Particularly relevant with security checks (e.g., signature or certificate validation)
  - Make your system create state information
  - (Make your system transmit data, preferably to somebody else)
- Issue: distinguishing legitimate work from attack
  - There is not necessarily a well-defined user behavior
  - Example HTTP: Botnet fetching pages, search crawlers, site replication (wget)
    - Some web sites inspect the HTTP `User-Agent:` header and deny access to bots
- General Approach
  - Avoid creating (much) state early on the server side
  - Make the client work harder (client-side easier to scale anyway)

Example: TCP SYN Cookies (1)

- Normal TCP operation when a SYN packet comes in
  - Create protocol control block (PCB), choose random initial sequence number, set cleanup timeout (in case you never get an ACK), send SYN-ACK back
  - State will last until timeout expires
- TCP SYN Cookie idea (D. J. Bernstein, 1996)
  - Do not create state
  - Encode the local state you would create in 32-bit sequence number
    - Part of this is protected cryptographically
  - Send SYN-ACK
  - If ACK comes back: recreate state from acknowledgement number
  - If no ACK comes back: nothing lost except for a few CPU cycles
  - Equalizes the burden
    - Attacker needs to respond and thus bind local resources
    - Attacker can no longer use random source addresses

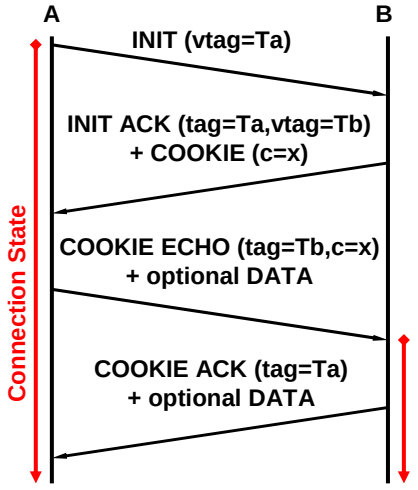
Example: TCP SYN Cookies (2)



- Issues: limits TCP option negotiation capabilities
  - E.g., large windows, SACK

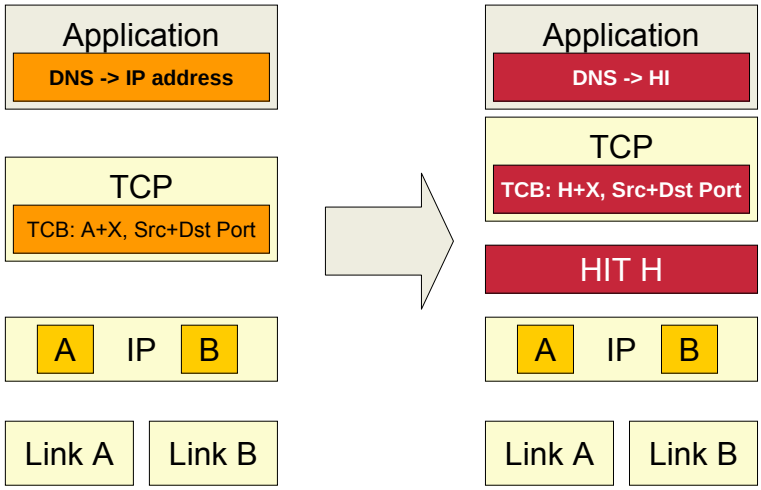
Example: SCTP Association Initialization

- A: choose verification tag Ta, send INIT
  - also: window size, TSN, other param.s
  - B uses Ta in all responses
- B: choose verification tag Tb, send ACK
  - returns local association state in cookie
    - e.g. state+check+lifetime+…+MAC(key,…)
  - do not keep local state
    - Cookie is variable length
- A: returns cookie to Tb
  - may include user data
- B: re-creates state from cookie
  - considers association established
  - responds with ACK
  - may include data as well

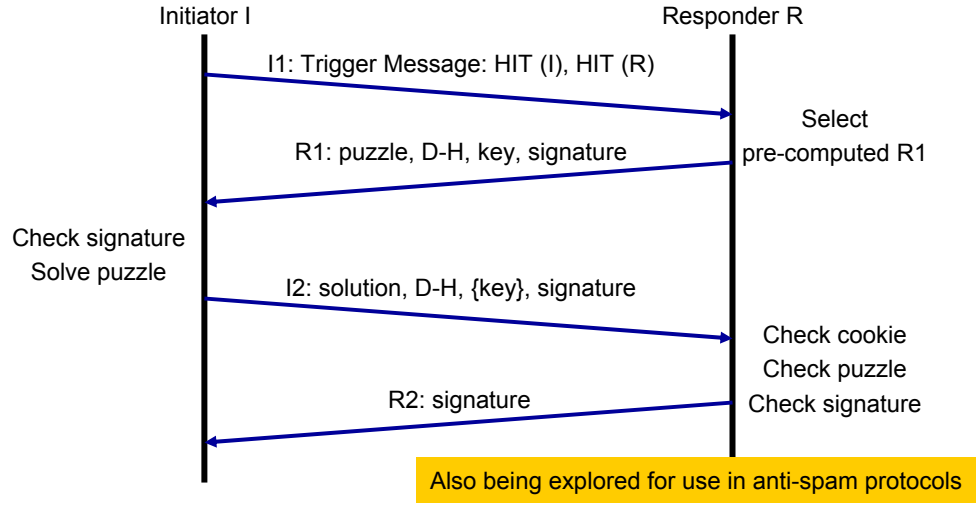


Related approaches taken for Internet Key Exchange (IKE) [RFC 4306], DCCP, DTLS.

Example: Puzzles in the Host Identity Protocol (HIP)



Example: Puzzles in the Host Identity Protocol (HIP)



Also being explored for use in anti-spam protocols

Example: Puzzles in the Host Identity Protocol (HIP)

- Puzzle example
  - Configurable complexity level K
    (chosen depending on assumed trust level of initiator)
  - Responder supplies a random number I (8 bytes)
  - Initiator must find a matching number J (8 bytes)
    - Compute SHA1 (Concatenate (I, HIT (Initiator), HIT (Responder), J))
    - So that the lowest order K bits of the result must be zero
  - Can only be done by repeatedly choosing J and trying
  - Responder can easily check by one-time calculation upon receipt of J
- CPU-bounded approach
  - Alternative: memory-bounded left for future study

Bad Example: HTTP Prefetching w/ MHTML

- Many small end-to-end interactions (GET – 200 OK) may slow down retrieving a web page over long delay links
- Idea 1: create some “wget -p” type of GET request
  - Shall return all resources of a web page
  - Send them multipart/related body (MHTML, RFC 2557)
  - Causes server load upon a single request
- Idea 2: Use transactional TCP (T/TCP)
  - T/TCP avoids initial 3-way handshake assuming that only a single message exchange will take place
  - Eliminates the protection available by means of TCP SYN cookies
  - Allows an attacker to use an arbitrary source address

Robustness Issues with Protocol Implementations

Implementation Robustness

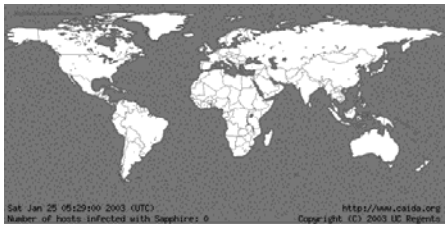
- Of course, your code shouldn't crash…
- But there is a deeper problem:
  - Most ways your code can be made to crash can be used for an attack
- Denial-of-Service Attacks
  - Crash
  - Loop
  - Performance problem
- Subversion of access control
  - Buffer overflows!

Buffer Overflows

- Most popular attack on server software
- Age-old problem (known since the 1960s)
- “Attack of the decade” (Bill Gates)
- Most Worms use Buffer Overflows (Morris Worm, Code Red, Blaster)
- Goal: smuggle in malicious code
- Based on your programming mistakes

SQL-Slammer/Sapphire

- 25 January 2003: Korea practically offline
- UDP Packet with 376 Bytes payload
  - Transmitted at maximum rate towards randomly chosen target IP addresses
- Warhol worm: infected most of the ≥ 75 000 victims within 10 minutes
  - Doubling every 8.5 seconds
  - Despite bug in PRNG



- Basis: Vulnerability in MSSQL server
  - Known since 24.07.2002
  - Attack targeted at MSSQL server
- Remedy: Close port 1434 (MSSQL)

Incorrect checks of input

```
char buf[42];
gets(buf);
```

- **Problem**:
  - C function `gets` does not check the length of the input ("unchecked buffer")
  - For input longer than 41 characters (null termination!): some memory around the variable "buf" is overwritten
  - Local Variables live on the Call Stack
- Solution: `fgets(buf,42,stdin)`

A local "unchecked buffer"

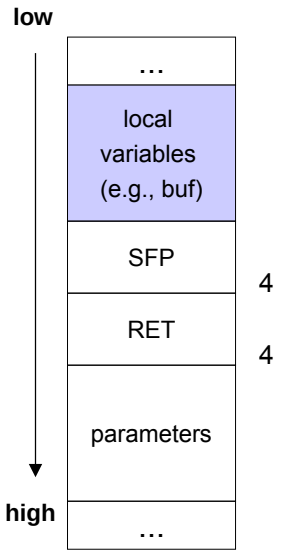
- UNIX Version 6 (ca. 1975), login program:

```
char user[100], passwd[100], correct[100];
gets(user); getpwnam(…); strcpy(correct, …);
gets(passwd);
if (strcmp(crypt(passwd, …),correct)) …
```

- Entering an 108 character password overwrites memory beyond the end of `passwd`, replacing the comparison value `correct` from the password file
- Fabricated password that encrypts into its own last 8 characters serves as master key!

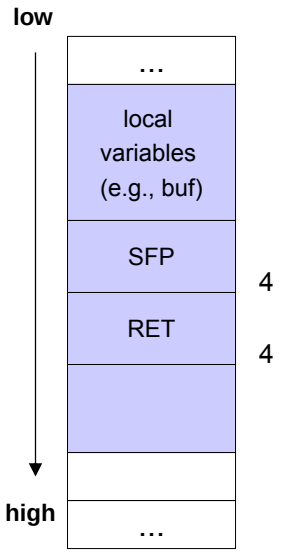
Call stack

- Local variables of C functions live on the call stack
- **Example**: Function call stack of the x86 CPU
- RET: return address
- SFP: Stack Frame Pointer



Overwriting the return address

- If the code can be tricked into writing beyond buf:
- Supply special input string that manipulates RET in such a way that the return jump leads into **exploit code**
- Easiest approach: put exploit code on the stack, too:
- Example for exploit code for Linux/Unix:
  `execl` to replace the running program with a Unix shell
- Shell then runs with privileges of server process and waits for input from the network
- Particularly useful if the server process had root rights



Preparing the input string

Start **End**

Padding	New return address (possibly repeated)	NOPs	Exploit code (e.g., shell)

Build a Landing Pad out of NOPs:

The *exact* address of the stack pointer may depend on hard-to-predict factors (e.g., total length of environment variables)

A little more work for the attacker

- Not enough space on the stack?
  - Machine code is **compact**
  - Many functions can be called, or simply use existing services:
    - E.g., Windows: many libraries (DLLs) are already linked into the process space of the victim server process
    - Can use API functions,
      e.g., *LoadLibrary*: get the libraries needed
- No data transparency in the protocol?
  - Just avoid characters like '\0' or '\n' — there are many ways to code the same function

Counter measure: NX-Bit

- Idea: Disallow running code on the stack
  - No way to do this in base x86 architecture
    - Extensions in current processors (AMD64: NX)
    - Windows: Data Execution Prevention (DEP)
  - Need to be careful with data areas that carry shared libraries
  - Need to allow code generation (JIT compilers!)
  - Some programs legitimately use the stack to run code
- Counter-counter measure: “Return” to library
  - Choosing the right parameters can have the same effect as your own code on the stack
  - But complex attacks get a bit harder to mount

Counter measure: Canaries

- Function prologue stores a special value (Canary) immediately besides the return address
- Function epilogue checks that the canary is still intact
  - If not: abort! (there is nothing to save…)
- Windows: Stack Cookies
- Disadvantage: a couple more instructions for a procedure call/return (may double cost)

Summary: Buffer-Overflows

- **Main cause for buffer overflows**: programming errors:
  use of predefined library functions in programming languages such as C or C++ without boundary checking:
  - strcpy(), strcat(), gets() in C
  - There are alternatives: strncpy(), strncat(), fgets().
- **Note**:
  Most OS services (Unix, Windows) are coded in C, C++.
  (e.g., Windows XP: some 40 M lines of C code —
  rough estimate: 5-50 bugs per 1000 lines of code!)
- Almost all worms exploit buffer overflows

SQL Injection

- Many web applications use a database backend to store persistent data (login information, customer addresses, …)
- Data is entered via web forms
- Web application languages such as PHP provide an easy-to-use database interface to move form input into the database
- Oops, again: Attacker defines the data!

... Attacker defines the data

Example:
Form field for entering an email address

PHP application might then use SQL statement:

```
SELECT email, passwd, login_id, name
FROM members
WHERE email='Data from the network'
```

Attacker enters:

```
bad@guy.fi'; DROP TABLE members;--
```

In SQL-Statement:

```
...WHERE email ='bad@guy.fi'; DROP TABLE members;--'
```


SQL injection: summary

- Again: non-validated input becomes program code
  - Variante: PHP- oder Perl-Code
- SQL injection attacks may not be easy:
  - Where to get the names of database relations and columns?
    - Error messages might have leaked those
  - Much easier just to create damage than to subvert access control
- Nonetheless: Bugtraq has news about new SQL injection attacks on a daily basis

Cross Site Scripting (XSS)

1. Web pages can contain scripts (e.g., JavaScript)
   - Scripts are executed on visitor's browser
   - Can access the Cookies the Website uses for authentication
2. Web pages can also contain user-defined data
   - E.g., based on previous input of a different user
   - E.g., based on a URL parameter

- Attacker can use (2) to foist a script on somebody else that is then executed during normal web page access (1)

A simple XSS attack

- **Example**:
  http://auction.example.com/filename.html returns an error message of the form:

  ```
  404 page does not exist: filename.html.
  ```

- Attacker might give victim a prepared link:
  http://auction.example.com/<script>alert('hello')</script>
- When the link is followed, the script within the link is executed on the browser of the victim
  - Script might compromise data in cookies

Summary: XSS attacks

- Attacker can run JavaScript on visitor's browser in the context of a different site (cross-site):
  - Read Cookies, store them elsewhere
  - Simulate password entry prompt
- Can be used for session hijacking
- Countermeasure: Website must validate all input (don't let unwanted scripts go through)
  - Unfortunately, there is half a dozen ways to provide scripting in HTML
  - In the end, the website must make sure only known HTML constructs make it from one user to another

Summary Implementation Robustness

- The root cause of Buffer Overflows, SQL Injection attacks and XSS attacks: unchecked input
- So:
  Check your input!
  Check your input!
  Don't trust that input!
- Don't just look for the known Problems
  - attackers have great new ideas all the time
- Only allow positively healthy input!

Can we win?

- Attacker needs only **one** security hole
- Defense must find **every** hole and fix it

- Real systems are too complex to be free of errors
- Intrusions need to be **prevented**, **detected**, **contained**

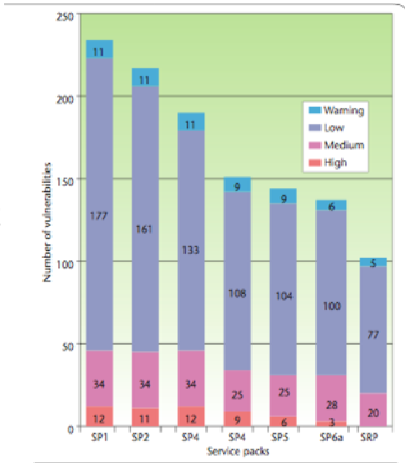


Figure 1. Security flaws. The number of security flaws in Windows NT 4.0 Service Packs 1 through 6a and the post-SP6a Security Rollup Package (SRP) according to risk levels.