Run-time Detection of Heap-based Overflows

William Robertson <wkr@cs.ucsb.edu>
Christopher Kruegel <chris@cs.ucsb.edu>
Darren Mutz <dhm@cs.ucsb.edu>
Fredrik Valeur <fredrik@cs.ucsb.edu>
UC Santa Barbara

Outline

- Motivation and Related Work
- Exploiting the Heap
- Heap Protection Technique
- Detection and Performance Evaluation
- Deployment
- Conclusions and Future Work

Motivation

- Why buffer overflow protection?
  - "Insecure" languages
  - Programmers are only human
- Why not use Java / C# / Cyclone / …?
- Why protect the heap?
- What solutions already exist?

Recent Heap Vulnerabilities

- OpenSSH < 3.7.1 buffer management vulnerability
- Snort stream4 preprocessor < 2.0 heap overflow vulnerability
- CVS < 1.11.5 double-free() vulnerability
- MS SQL server resolution service heap overflow vulnerability
- …

Related Work

- Automatic buffer bounds checking
  - gcc bounds-checking patch [Jones, Kelly]
- Preventing stack-based overflows
  - ProPolice [Hiroaki Etoh et al.]
  - StackGuard [Crispin Cowan et al.]
  - StackShield [Vendicator]
  - Libsafe / Libverify [Baratloo, Singh, Tsai]

Related Work (cont.)

- Preventing execution on the stack
  - Linux non-exec stack [Solar Designer]
- Preventing execution on the heap
  - PAX
- Memory protection systems
  - Valgrind [Julian Seward]
  - Electric Fence [Bruce Perens]

Outline

- Motivation and Related Work
- Exploiting the Heap
- Heap Protection Technique
- Detection and Performance Evaluation
- Deployment
- Conclusions and Future Work

The GNU C Library Heap

- Based on Doug Lea's dlmalloc
- Uses boundary tags and binning
- Memory allocated in chunks
  - In-band management information (boundary tag)
  - Application-usable memory region
- Free chunks kept in bins

glibc memory chunks

```
struct malloc_chunk
{
  INTERNAL_SIZE_T prev_size;
  INTERNAL_SIZE_T size;
  struct malloc_chunk *bk;
  struct malloc_chunk *fd;
};
```

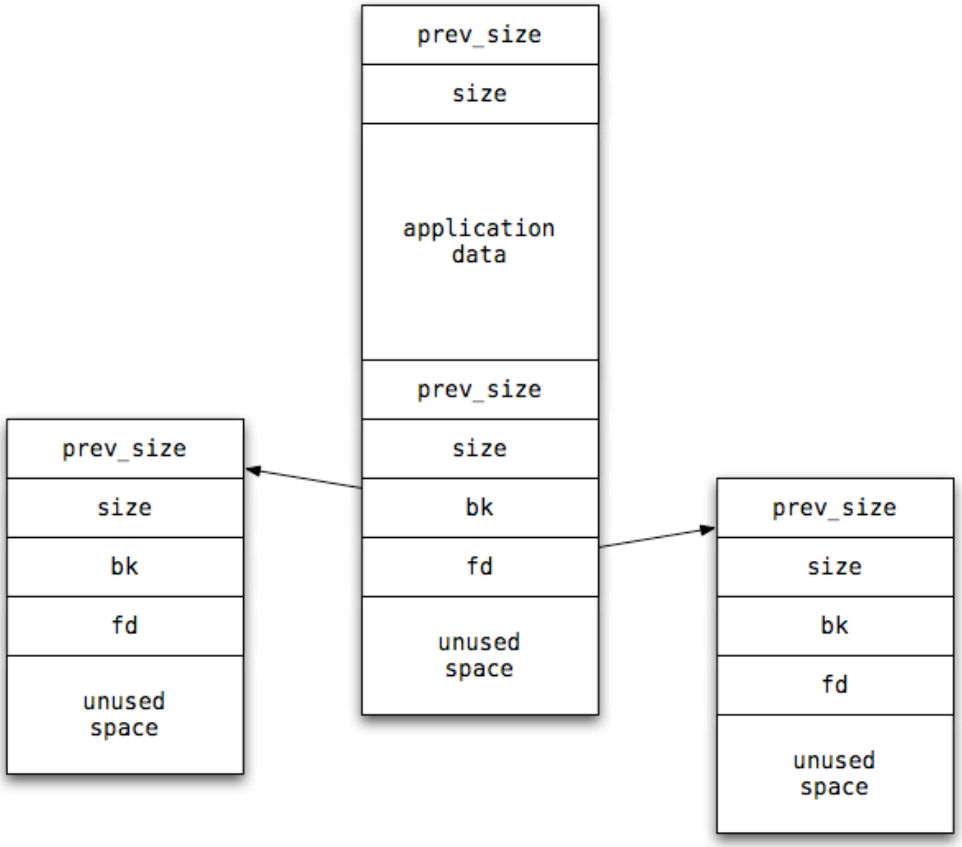
Exploiting Heap Overflows

- Heap buffer overflow overwrites header of next chunk in memory
- Attacker controls values placed in overflown chunk header
- Heap management routines tricked into writing controlled value into chosen memory location

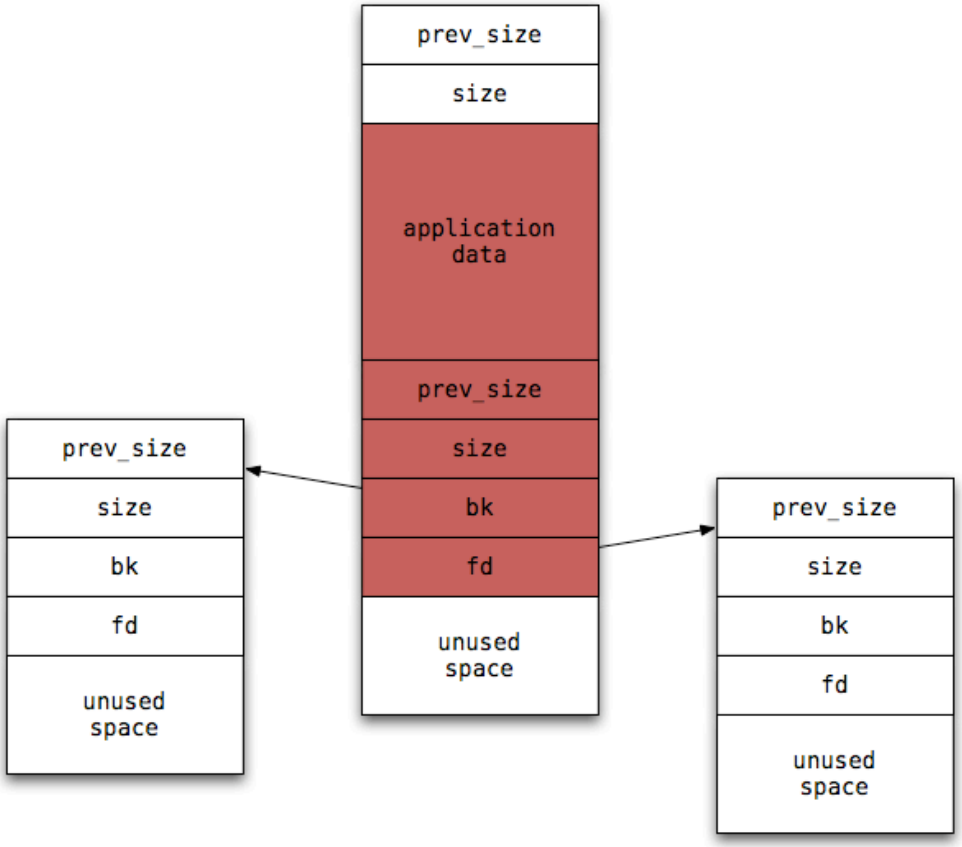
unlink()

```
#define unlink(P, BK, FD) { \
  FD = P->fd;               \
  BK = P->bk;               \
  FD->bk = BK;              \
  BK->fd = FD;              \
}
```

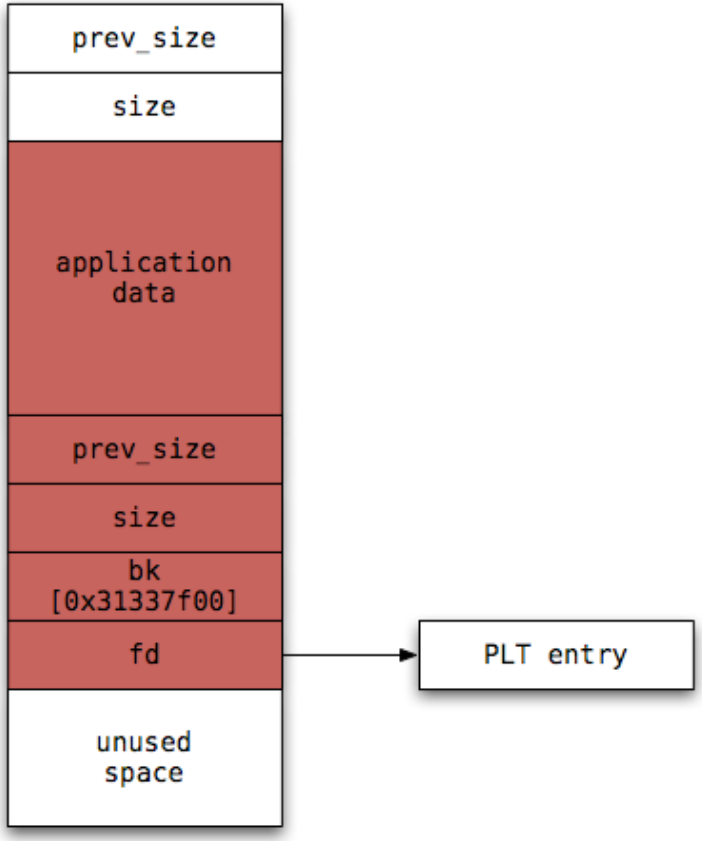
Heap Overflow



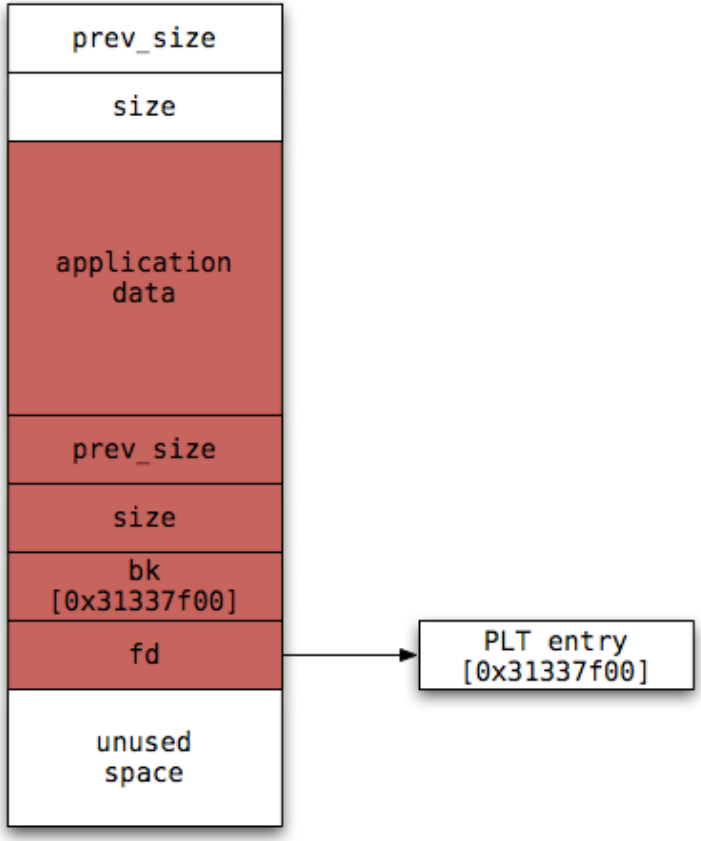
Heap Overflow (cont.)



Heap Overflow (cont.)



Heap Overflow (cont.)



Exploit Variants

- Heap overflow exploit variants
  - frontlink() macro
  - Fake chunk headers, size field manipulation
- Variations of basic exploit
  - Can be handled with one defensive technique

Outline

- Motivation and Related Work
- Exploiting the Heap
- Heap Protection Technique
- Detection and Performance Evaluation
- Deployment
- Conclusions and Future Work

Heap Protection Technique

- Adaptation of canary-based stack protection schemes
- Preface memory chunks with seeded checksum of header fields
- Check integrity of header before performing operations upon it

Heap Protection (cont.)

- Canaries seeded with random number
- What prevents attacker from setting seed to known value?
  - Random seed protected with mprotect ()
  - Costly, but only performed once per process

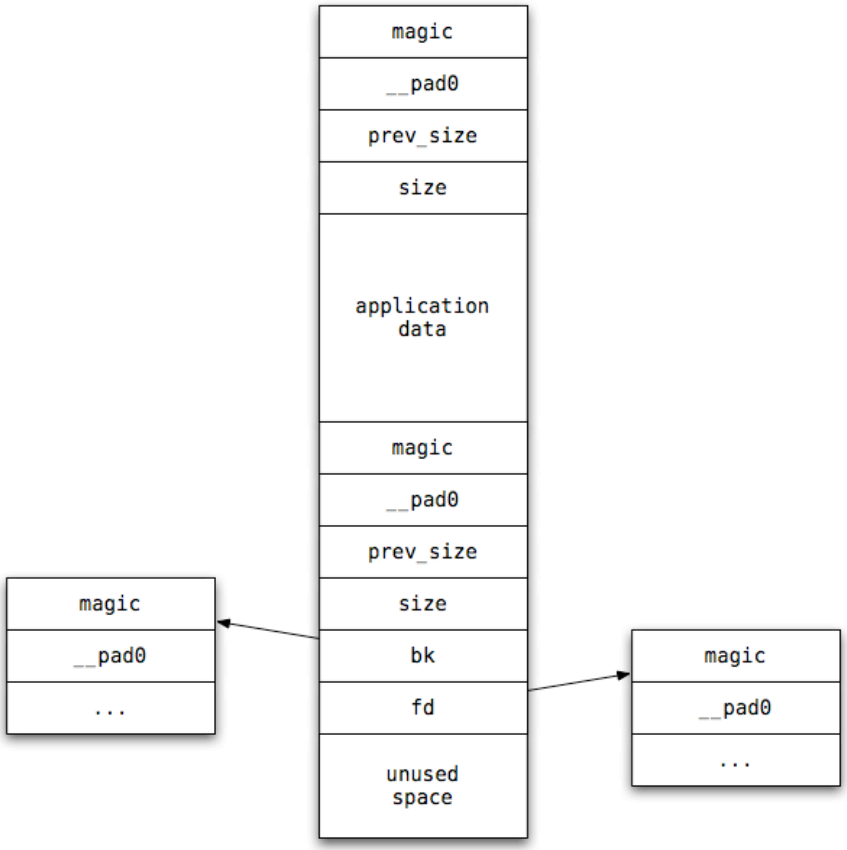
Modified glibc memory chunks

```
struct malloc_chunk
{
  INTERNAL_SIZE_T magic;
  INTERNAL_SIZE_T __pad0;
  INTERNAL_SIZE_T prev_size;
  INTERNAL_SIZE_T size;
  struct malloc_chunk *bk;
  struct malloc_chunk *fd;
};
```

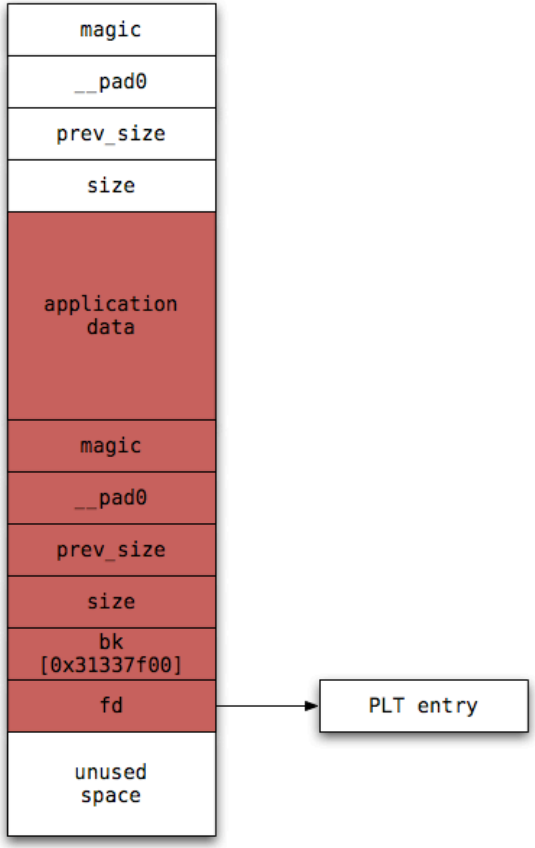
Heap Overflows Reloaded

- Heap buffer overflow overwrites header of next chunk in memory, overwriting next chunk header's canary
- Attacker controls values placed in overflown chunk header
- Chunk header integrity check detects overflow has occurred, process aborts

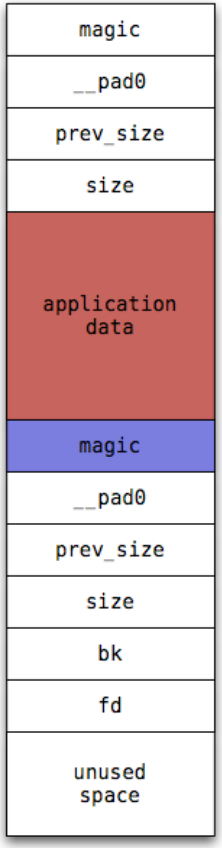
Overflow Detected



Overflow Detected (cont.)



Overflow Detected (cont.)



Outline

- Motivation and Related Work
- Exploiting the Heap
- Heap Protection Technique
- Detection and Performance Evaluation
- Deployment
- Conclusions and Future Work

Evaluation Goals

- Demonstrate detection capability
- Demonstrate low impact on application performance
- Demonstrate system stability
- Superiority over existing glibc debugging code

Detection Evaluation

- Ran several recent heap-based exploits against a test system
- Test system configured in three states
  - no protection
  - glibc debugging enabled
  - glibc with heap protection enabled

Detection Evaluation Results

Exploit	glibc	glibc + debugging	glibc + heap protection
wu-ftpd	shell	aborted	aborted
sudo	shell	aborted	aborted
cvs	segfault	aborted	aborted
unlink	shell	aborted	aborted
frontlink	shell	aborted	aborted
evasion	shell	shell	aborted

Performance Evaluation

- Micro-benchmarks
  - Tight loop of randomly-sized allocations
  - AIM9 memory benchmark
- Macro-benchmarks
  - OSDB (PostgreSQL 7.2.3)
  - WebStone (Apache 2.0.40)

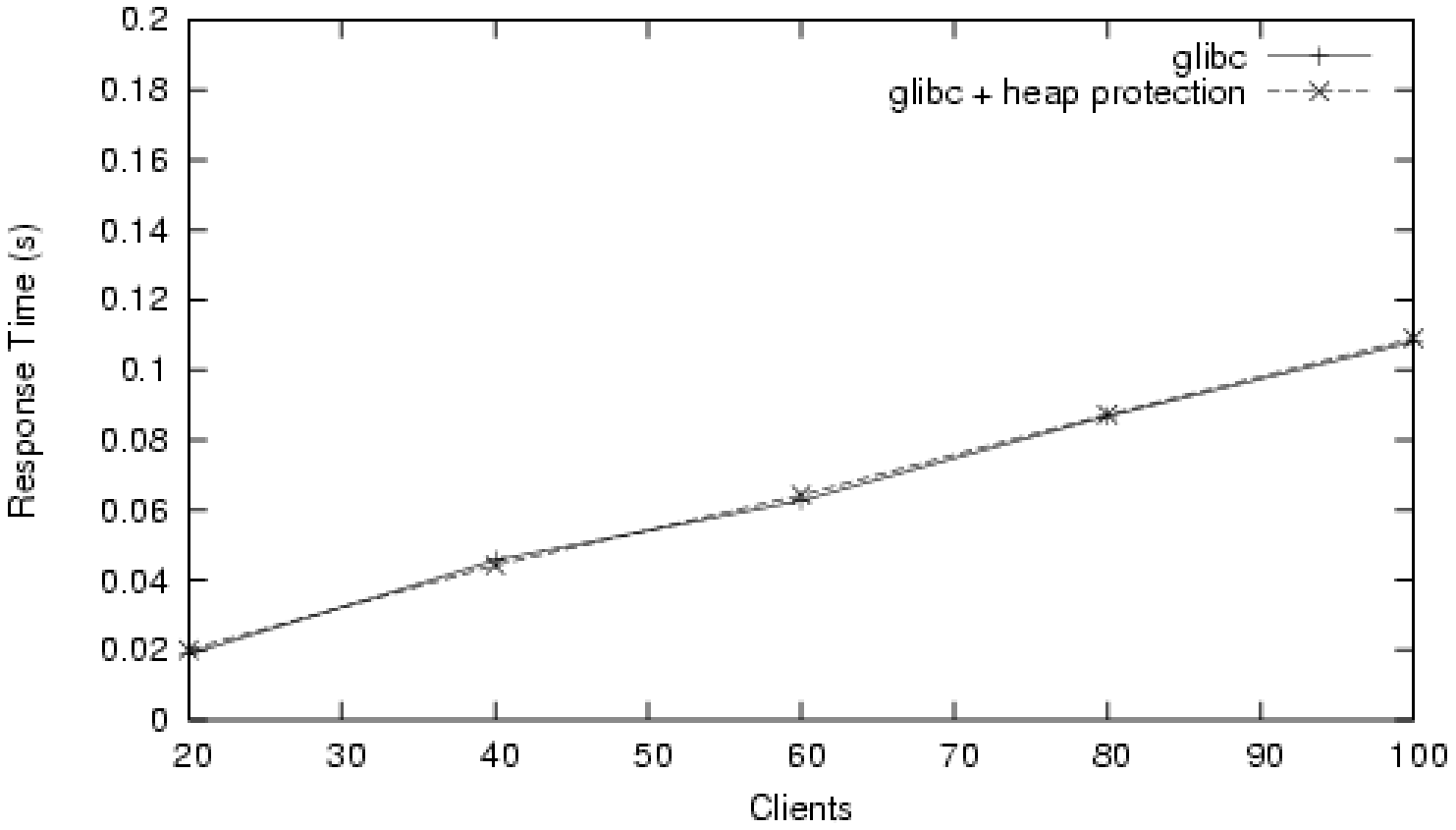
Micro-benchmark Results

Benchmark	glibc	glibc + debugging	glibc + heap protection
Loop	1,587 s	2,621 s (+65%)	2,033 s (+28%)
AIM9	5,094 s	7,603 s (+49%)	5,338 s (+05%)

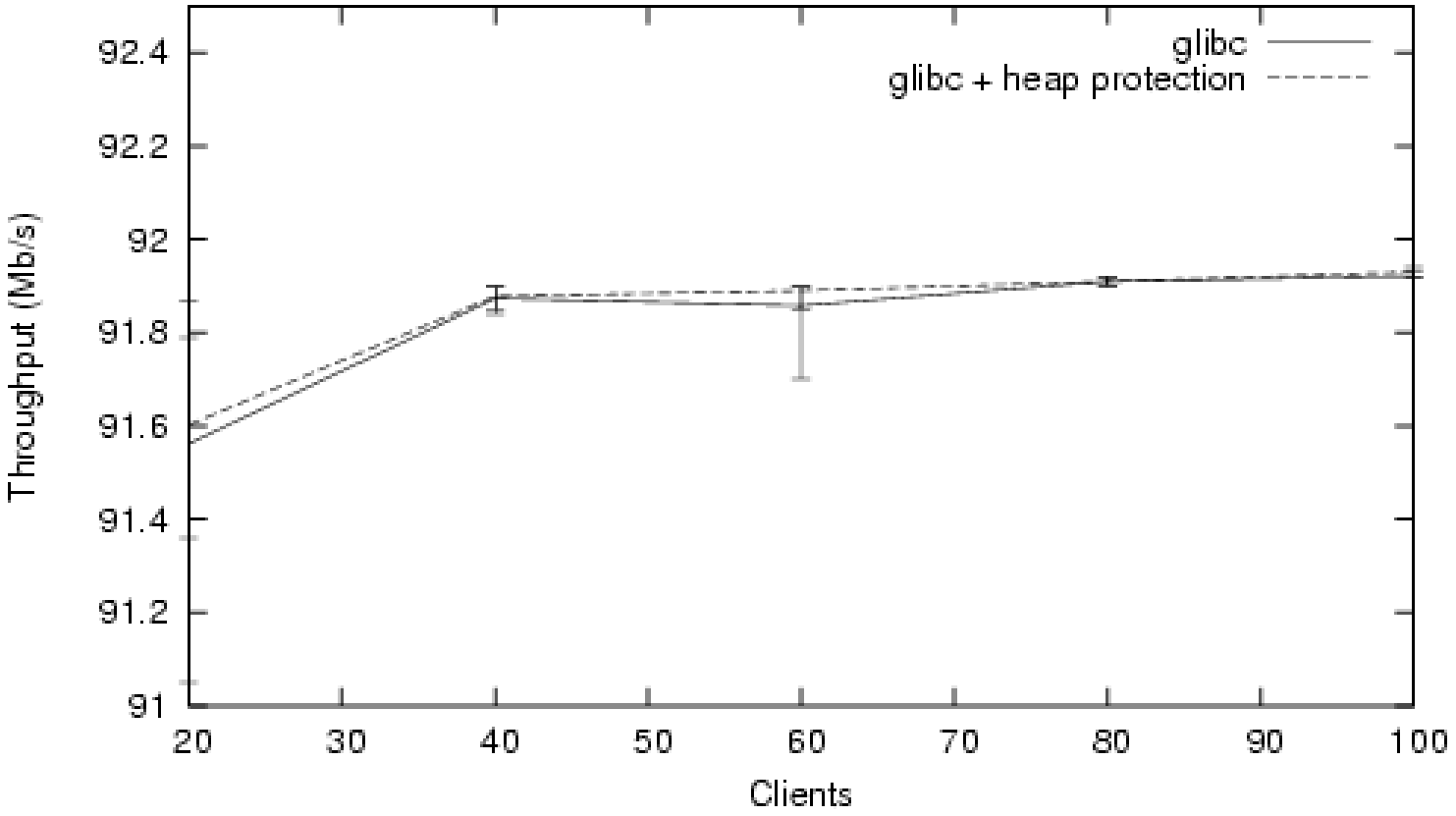
OSDB Benchmark Results

Benchmark	glibc	glibc + heap protection
OSDB	6,015 s	6,070 s (+0.91%)

WebStone Results (response)



WebStone Results (throughput)



Stability Evaluation

- Ran memory-intensive applications on protected test system for period of four weeks
- Deployed on exposed lab machines, desktops of several authors
- No crashes or other known issues at this time

Outline

- Motivation and Related Work
- Exploiting the Heap
- Heap Protection Technique
- Detection and Performance Evaluation
- Deployment
- Conclusions and Future Work

Deployment

- System-wide protection
  - all applications using glibc's heap automatically protected
- Per-application protection
  - uses system loader's LD_PRELOAD
  - minimize system performance hit
  - minimize impact of any stability issues

Deployment (cont.)

- Available as glibc patch
- Binary packages available for selected operating systems and architectures

Conclusions

- Effective detection and prevention of heap-based exploits
- Low performance impact in most cases
- Transparent to existing applications
- Simple to deploy
- Necessary component of layered defense against system compromise

Future Work

- Adapt technique to similar heap management systems
- http://www.cs.ucsb.edu/~rsg/heap