CS590-SWS/527 Software Security

Memory Safety

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Kyriakos Ispoglou

https://nebelwelt.net/teaching/16-527-SoftSec/

Spring 2016

Table of Contents

1. Eternal War in Memory
2. Memory safety
   - Spatial memory safety
   - Temporal memory safety
   - Towards a definition
3. SoftBound
4. CETS
5. Summary and conclusion

The Eternal War in Memory

- Memory corruption is as old as operating systems and networks.
- First viruses, worms, and trojan horses appeared with the creation of operating systems.
- All malware abuses a security violation, either user-based, hardware-based, or software-based.
- Early malware tricked users into running code (e.g., as part of the boot process on a floppy disk).
- The Morris worm abused stack-based buffer overflows in sendmail, finger, and rsh/exec to gain code execution on a large part of the Internet in 1988.
- A plethora of other software vulnerabilities continued to allow malware writers to gain code execution on systems.

Control-flow hijack attack

The attacker wants to control the execution of a program by redirecting control-flow to an attacker-controlled location.

- First, the attacker must overwrite a code pointer (return instruction pointer on the stack, target of an indirect jump, or target of an indirect call).
- Second, the attacker forces the program to follow the modified code pointer (this can be explicit or implicit).
- Third, the attacker keeps modifying code pointers so that the process now executes the attacker's code.

Control-flow hijack attack

- Memory Safety
- Integrity
- Randomization
- Flow Integrity
- Successful Attack!

```
1 void vuln(char *u1) {
2   /* assert(strlen(u1) < MAX); */
3   char tmp[MAX];
4   strcpy(tmp, u1);
5   return strcmp(tmp, "foo");
6 }
7 vuln(exploit);
```

Quick attack overview

- Code corruption.
- Code injection and control-flow hijacking.
- Code reuse and control-flow hijacking.
- Control-flow bending.
- Controlling a Turing-complete interpreter inside an application (constrained execution).

Table of Contents

1 Eternal War in Memory

2 Memory safety
- Spatial memory safety
- Temporal memory safety
- Towards a definition

3 SoftBound

4 CETS

5 Summary and conclusion

Memory safety

Definition: Memory safety

Memory safety is a property that ensures that all memory accesses adhere to the semantics defined by the source programming language. The gap between the operational semantics of the programming language and the underlying instructions provided by the hardware allow an attacker to step out of the restrictions imposed by the programming language and access memory out of context. Memory unsafe languages like C/C++ do not enforce memory safety and data accesses can occur through stale/illegal pointers.

Memory safety

- Memory safety is a general property that can apply to a program, a runtime environment, or a programming language[1].
- A program is memory safe, if all possible *executions* of that program are memory safe.
- A runtime environment is memory safe, if all possible programs that can run in the environment are memory safe.
- A programming language is memory safe, if all possible programs that can be expressed in that language are memory safe.
- If memory safety is enforced, then a list of bad things can never happen. This list includes buffer overflows, NULL pointer dereferences, use after free, use of uninitialized memory, or illegal frees.

[1] See Mike Hicks definition of memory safety `http://www.pl-enthusiast.net/2014/07/21/memory-safety/`.

Memory safety

Memory safety violations rely on two conditions that must both be fulfilled:

- A pointer either goes out of bounds or becomes dangling.
- This pointer is used to either read or write.

Spatial memory safety

Definition: Spatial memory safety

Spatial memory safety is a property that ensures that all memory dereferences are within bounds of their pointer's valid objects. An object's bounds are defined when the object is allocated. Any computed pointer to that object inherits the bounds of the object. Any pointer arithmetic can only result in a pointer inside the same object. Pointers that point outside of their associated object may not be dereferenced. Dereferencing such illegal pointers results in a spatial memory safety error and undefined behavior.

Spatial memory safety

```
1 char *ptr = malloc(24);
2 for (int i = 0; i < 26; ++i) {
3   ptr[i] = i+0x41;
4 }
```

Temporal memory safety

Definition: Temporal memory safety

Temporal memory safety is a property that ensures that all memory dereferences are valid at the time of the dereference, i.e., the pointed-to object is the same as when the pointer was created. When an object is freed, the underlying memory is no longer associated to the object and the pointer is no longer valid. Dereferencing such an invalid pointer results in a temporal memory safety error and undefined behavior.

Temporal memory safety

```
1 char *ptr = malloc(24);
2 free(ptr);
3 for (int i = 0; i < 24; ++i) {
4   ptr[i] = i+0x41;
5 }
```

Memory safety

- Assume we have defined (allocated) and undefined memory. Assume that deallocated memory is never reused. Memory safety is violated if undefined memory is accessed.
- Pointers can be seen as capabilities, they allow access to a certain region of (allocated) memory. Each pointer consists of the pointer itself, a base pointer, and bounds for the pointer.
- When creating a pointer, capabilities are initialized. When updating a pointer, the base and bounds remain the same. When copying a pointer, the capabilities are propagated. When dereferencing a pointer, the capabilities are checked.
- Capabilities cannot be forged or constructed by the programmer but are inherently added and enforced by the compiler.

Memory safety

- Capability-based memory safety is a form of type safety with two types: pointer-types and scalars.
- Pointers (and their capabilities) are only created in a safe way. The capabilities are created as a side-effect of creating the pointer.
- Pointers can only be dereferenced if they point to their assigned memory region and that region is still valid.

Memory safety enforcement

We know how memory safety violations look like. How can we protect against them?

- Verification?
- Testing?
- Fuzzing?
- Symbolic Execution?
- Yes, these are all correct but will either not find all bugs, are not complete, or have high overhead.
- We need to enforce memory safety at runtime by checking the security property at strategic locations.

Bounds checking approaches in C/C++

Tripwire: use few bits of state for each byte of memory, place red-zones between blocks (e.g., Valgrind's memory checker, Google's AddressSanitizer)

Pointer based: fat pointers are used to check dereferences (e.g., Cyclone, CCured)

Object based: must point within same object, check pointer manipulations (e.g., SafeCode)

All mechanisms are challenged by (i) high runtime overheads, (ii) incompleteness handling casts, (iii) incompatible pointer representations (e.g., syscalls), (iv) code incompatibilities.

Table of Contents

1 Eternal War in Memory

2 Memory safety
- Spatial memory safety
- Temporal memory safety
- Towards a definition

3 SoftBound

4 CETS

5 Summary and conclusion

SoftBound

- Compiler-based transformation that enforces spatial memory safety[2]
- No source code changes necessary, compiler-based
- No memory layout changes necessary, disjoint meta data (fat pointers)
- Simple, intra-procedural analysis
- Effective, no false positives/negatives
- Low(-ish) overhead of 67% for SPEC CPU2006

[2]SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. Santosh Nagarakatte et al, PLDI'09.

Alternative approaches

Object based: Cannot detect sub-object overflows, overhead for range lookups. But meta data is disjoint, therefore high compatibility.

Fat pointers: Low compatibility through inline meta data. Can detect sub-object overflows.

Both fail to protect against arbitrary casts (across incompatible types). Notable exception is CCured with pointer classification.

SoftBound approach

```
1  struct BankAccount {
2    char acctID[3]; int balance;
3  } b;
4  b.balance = 0;
5  char *id = &(b.acctID);
6  lookup(&id)->bse = &(b.acctID);
7  lookup(&id)->bnd = &(b.acctID)+3;
8  char *p = id;  // local, remains in register
9  char *p_bse = lookup(&id)->bse;
10 char *p_bnd = lookup(&id)->bnd;
11 do {
12   char ch = readchar();
13   check(p, p_bse, p_bnd);
14   *p = ch;
15   p++;
16 } while (ch);
```

SoftBound instrumentation

- Initialize (disjoint) metadata for pointer when it is assigned.
- Assignment covers both creation of pointers and propagation.
- Check bounds whenever pointer is dereferenced.

```
1 if (p < p_bse) abort();
2 if (p + size > p_pnd) abort();
3 value = *p
```

Check results in five x86 instructions: `cmp`, `br`, `add`, `cmp`, `br`.

SoftBound details

- Many small nits requires so that it works out.
- Separate compilation and library code are issues.
- Function pointers and variadic arguments are problematic.
- `memcpy` needs to be handled in a special way.
- Illegal (but "working") casts can be problematic.

Table of Contents

1 Eternal War in Memory

2 Memory safety
- Spatial memory safety
- Temporal memory safety
- Towards a definition

3 SoftBound

4 CETS

5 Summary and conclusion

Compiler-Enforced Temporal Safety for C

- Temporal memory safety is an orthogonal problem to spatial memory safety.
- The same memory area can be allocated to new object.

Heap-based temporal safety error

```
1 int *p, *q, *r;
2 p = malloc(8);
3 ...
4 q = p;
5 ...
6 free(p);
7 r = malloc(8);
8 ...
9 ... = *q;
```

Stack-based temporal safety error

```
1 int *q;
2 void foo() {
3   int a;
4   q = &a;
5 }
6 int main() {
7   foo();
8   ... = *q;
9 }
```

Compiler-Enforced Temporal Safety (CETS) for C

- How do you ensure that a pointer references the new object and not the old object? How do you detect stale pointers?
- One solution is garbage collection (free is noop but periodically scan for unused memory areas with no pointers to them).
- Another is to not reuse memory.
- A third is to use versioning. Each allocated memory object and pointer is assigned a unique version. Upon dereference, check if the pointer version is equal to the version of the memory object. Two failure conditions: area was deallocated and version is smaller (0) or area was reallocated to new object and the version is bigger.

Existing approaches and drawbacks

- Simple approaches track validity based on the location of memory objects. Unfortunately, these approaches cannot distinguish between a reference to a reallocated object and the original object.
- SafeC used a global set for memory objects and fat pointers to store validity data with the drawback of high overhead due to the set lookups.
- Follow up work introduced specific lock addresses and lock tables that stored validity information, removing the need for a potentially high overhead hash lookup to find the target address.

CETS mechanism

1. Identify all memory allocation functions and assign a unique version to the memory area. Assign the same version to the initial pointer that is returned from the allocation function.
2. Identify all memory deallocation functions and destroy (zero out) the version of the associated memory area.
3. When a pointer is assigned, propagate the version from the rhs.
4. When a pointer is dereferenced, check if the version of the pointer and pointed-to object match.

Table of Contents

1 Eternal War in Memory

2 Memory safety
- Spatial memory safety
- Temporal memory safety
- Towards a definition

3 SoftBound

4 CETS

5 Summary and conclusion

Summary

- Memory safety has been an issue for generations of programmers and has severe security implications.
- Distinguish between spatial and temporal memory safety violations.
- SoftBound protects against spatial memory safety errors, CETS against temporal memory safety errors.
- Reading assignments:
  - Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. IEEE S&P'13. `http://nebelwelt.net/publications/files/13Oakland.pdf`.
  - Santosh Nagarakatte, Milo M. K. Martin, Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. `http://drops.dagstuhl.de/opus/volltexte/2015/5026/pdf/16.pdf`.

Questions?

?