

Source Analysis to Binary Analysis

Jeff Perkins

MIT CSAIL

Requirements for learning

- Program points (places in the code to look for constraints)
- Variable definitions
 - program point
 - type (pointers, integers, strings, arrays, etc)
- Values
 - Each time program point is executed.
 - Valid (initialized and allocated) data only
 - uninitialized values will corrupt learning
 - uninitialized values will cause false positives during checking
 - following deallocated or uninitialized pointers can crash the program
- Overhead should be less than 2x

Source Analysis

- Program Points
 - Method enter/exit
 - Generalized to classes
- Variables
 - Visible external source variables (parameters, class variables, globals)
 - Fields are followed to a specified depth (e.g., $a \rightarrow f1 \rightarrow f2$)
- Value validity
 - calls to allocate/deallocate memory are intercepted
 - Each memory read/write is checked
 - Only initialized values in allocated memory are considered
 - Overhead is $\sim 100x$
- Can be implemented by source transformation or by dynamic analysis utilizing debug information.

Binary analysis - source inference

- Source/debug information is not available
- Infer source information
- Track each read/write to find structure definitions
- Track stack accesses to find function parameters
- Determine value validity by tracking read/writes and intercepting allocation/deallocation calls (as in the source analysis)

Problems with source inference

- Allocation methods are not known
 - Many windows programs use custom allocators
 - Stale pointers will be difficult to detect.
 - Code within custom allocators may confuse inference
- Machine code is irregular
 - Frame pointer is not always used
 - Compilers obfuscate code
 - Programs include hand written assembly code
 - Some code is self-modifying
 - Assumptions about code are *always* wrong
- Overhead is high
 - Value validity analysis cannot be partitioned to reduce overhead
 - Difficult to partition heap variable inference

Binary analysis - binary variables

- Don't try to reconstruct source information
- Binary variables are machine code elements (registers/memory at a program offset) that the program manipulates directly
 - `036FEE:ecx`
 - `036FF0:04000`
- Basic types can be determined from the opcode
 - Elements used as an address are pointers
 - Elements that are manipulated numerically (add, subtract, etc) are integers.
- Parameters to standard calls (e.g., `strcpy`) can also be variables
 - debug information is available for standard libraries
 - aggregate types (such as strings can be determined)
- Comparability information can also be inferred

Example

```
struct REQ {  
    int buf_size;  
    int max_buf;  
    int fl;  
    char *string;  
};
```

```
static char *handle_req (struct REQ *req) {  
    if (req.buf_size > req.max_buf) {  
        return NULL;  
        ...  
    }  
}
```

```
01  push    ebp                // push previous frame pointer on stack  
02  mov     ebp, esp           // set current frame at top of stack  
03  mov     eax, [ebp+8]       // load the address of arg0 into eax  
04  mov     ecx, [ebp+8]       // load the address of arg0 into ecx  
05  mov     edx, [eax]         // load buf_size into edx  
06  cmp     edx, [ecx+4]       // load max_buf into edx.  
07  jle    short loc_11        // jmp if less than or equal  
08  xor     eax, eax           // set eax to 0  
09  pop     ebp               // restore the previous frame pointer  
10  retn
```

Example

```
struct REQ {  
    int buf_size;  
    int max_buf;  
    int fl;  
    char *string;  
};
```

```
static char *handle_req (struct REQ *req) {  
    if (req.buf_size > req.max_buf) {  
        return NULL;  
        ...  
    }  
}
```

```
01  push    ebp                // push previous frame pointer on stack  
02  mov     ebp, esp          // set current frame at top of stack  
03  mov     eax, [ebp+8]      // load the address of arg0 into eax  
04  mov     ecx, [ebp+8]      // load the address of arg0 into ecx  
05  mov     edx, [eax]        // load buf_size into edx  
06  cmp     edx, [ecx+4]      // load max_buf into edx.  
07  jle    short loc_11       // jmp if less then or equal  
08  xor     eax, eax          // set eax to 0  
09  pop     ebp              // restore the previous frame pointer  
10  retn
```

Variable Information

```
01  push    ebp                // push previous frame pointer on stack
02  mov     ebp, esp           // set current frame at top of stack
03  mov     eax, [ebp+8]       // load the address of arg0 into eax
04  mov     ecx, [ebp+8]       // load the address of arg0 into ecx
05  mov     edx, [eax]         // load buf_size into edx
06  cmp     edx, [ecx+4]      // load max_buf into edx.
07  jle    short loc_11       // jmp if less then or equal
08  xor     eax, eax           // set eax to 0
09  pop     ebp                // restore the previous frame pointer
10  retn
```

- 03: ebp is a pointer
- 04: ebp is a pointer
- 05: eax is a pointer
- 06: ecx is a pointer
- 06: edx and 06: [ecx+4] are integers (compared, not dereferenced)

Advantages of binary variables

- Source information is not needed.
- Obfuscated or complex machine code constructs are less of a problem
- All binary variables contain valid values
 - Only machine code elements that are manipulated by the program are used
 - If the program uses uninitialized values, using them for learning/checking is reasonable.
- Variables critical to security are included
 - Addresses and indices used in indirect reads/writes
 - Strings
- Overhead is reasonable and can be partitioned in the community