POLITEHNICA University of Bucharest
Automatic Control and Computer Science Faculty

DWARFS & ELFS FOR STATIC BINARY CHECKING

descheck
DWARF & ELF Static Binary Checker

Lucian Adrian Grijincu
Codrin Alexandru Grajdeanu

Binary code analysis vs. source code analysis

Source code analysis is difficult because of several complex and hard to solve issues. A source code checker must implement a full standard C/C++ parser, but it also must be able to understand compiler-specific extensions. Many C programs use extensions like GCC's asm for inline assembly (every system call in glibc on Linux is implemented as an *asm* block that generates a software interrupt) or *__attribute__* for adjusting alignment, inter-positioning, thread specific variables (such is the case of the very common errno). Furthermore some compilers understand several *#pragma* directives which may alter the generated code. This is specific to many system header files like *<stdio.h>*, so simple 'Hello, world' applications may use compiler-specific extensions. In the best case scenario, building a static code checker that conforms only to ANSI-C or C99 or whatelse standard may not detect some bugs in the analysed files; in the worst case, it may render the source code uncheckable at all. Given the number of compilers, compiler versions and of different extensions each have, the parts of the C standards that are intentionally left ambiguous, categorized into implementation-defined, unspecified and undefined we see this as a too great challenge put in front of source analysis. Because programs may be dependent on undefined behaviour, a static code checker has to know which is the compiler that shall be used, but even so it is quite difficult to have static source code checker since unspecified and undefined behaviours are not required to be documented. Furthermore this tool must know beforehand all the *#defines* that will be used for each file.

Static code analysis vs. dynamic code analysis

The problem with dynamic code analysis s that it will only check some paths of the binary code – those activated at that specific run of the code. Heissenberg's uncertainty principle sometimes is applicable in software too: checking the code may change something in the overall environment of the application (a simple example: an application that is under tests will run a longer period of time, and time dependent paths of code may be altered, signal-handlers may interrupt different functions, etc.). Dynamic code analysis is even harder in the case of multi threaded applications: the behaviour of the application may be different from run to run. Given the fact that these tools are addressed to developers and quality-assurance testers, having them output different diagnostics at every run may be somewhat annoying.

All these problems instruct us to choose a different path: check the binary code statically. This way we may check more code paths than dynamic analysis, while not being enforced to have extensions for each compiler's quirks. Also output should be identical for two checks of the same input binary. Even more, we may detect all kinds of predefined common errors, and may be able to instruct the users in a more detailed fashion which was their mistake and what could be done to solve that issue.
One thing must be kept in sight: one cannot proof-check an application. One cannot even tell (in general) whether a program will ever terminate (successfully ore by a segfault/coredump, etc.). So trying to simulate the program considering different inputs to check as many paths as possible is not feasible, although simple simulations inside functions and for small function call chains may seem suitable.
All things considered, we cannot ignore the importance of checking the program dynamically (with the help of tools such as valgrind, electric fence, etc.) or that of writing solid test cases and most importantly thinking clearly about the program being written (and not relying in the checker to do the thinking for us). These tools are here to check on things one might miss out when thinking about "the big picture".

Implementation Details

We will present further some of the issues that such a tool may analyse and detect and some possible implementations for them but lets see the building blocks on which they'll be implemented.

ELF

Executable and Linkable Format is a common standard file format for executables, object code, shared libraries, and core dumps. The binaries that we'll analyse are in this format and it's de default format for output in gcc under most unices. These files consist of a beginning header followed by zero or more segments and zero or more sections. The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking, relocation and debugging. Because compiled code is extremely hard to read and giving users information about which assembly instruction mixed up is of little or no use (a usual developer or a qa tester will not understand such a terse and meaningless message and will not use this tool much, if at all), we will only analyse elf object files that were created with debugging sections (more exactly with DWARF2 debugging information).

The Binutils GNU package (found in many GNU/Linux distributions and implementations) will be used in several places. It provides a *objdump* application that we will modify to get the debugging information in a suitable format and the *libelf* library which will be used to read the original information from the object files. A second feature is the presence of a disassembling function that understands several processor architectures. We will use this to step through the code, simulating it's behaviour and checking the binary for errors.

DWARF

It's a standard debugging information format available for most of the major unices. It provides detailed information about types of variables, definition locations, references to source code (so that the original code may be extracted and shown to the user of the library – a much more useful piece of information that the assembly instruction that mixed something up), it also covers functions, structures and unions, typedefs and other items, being able to tell the size of any data type, the parameters of all functions defined in the object, etc.

This data is located in *.debug_info, .debug_aranges, .debug_abbrev* sections of the analyzed binary. To read this data we could just use the *libdwarf* library which is also found on many unices (including GNU/Linux distributions and Solaris). The problem with this is that traversing through it is fairly hard. So reading all the data at once and storing it in a useful manner is more appropriate. For this we use the code called when one runs *objdump –dwarf* / *dwarfdump* (depending on which of those is available on the system).

A general outline of this approach would imply these steps:

1. locate all the segments and sections of the binary (with libelf)
2. parse the dwarf2 debugging sections and store the information in memory as appropriate (as lists of function names, prototypes, type definitions, global, static, automatic variable definitions, etc.)
3. disassemble the code and check each instruction against some predefined rules.
4. process the data gathered at the last step and determine possible flaws in the analyzed binary file
5. build a report containing useful information (source file and line, the cause of the problem and a possible solution where suited) and output it to the user.

Case study: *defcall*

a DWARF & ELF function call graph viewer

1. locate all the segments and sections of the binary (with libelf)
2. parse the dwarf2 debugging sections
   - create a list of functions defined in the binary (*function_table)*
   - read with libelf all the dynamic symbols (which are stored in a special section) and determine which library functions are being called (wrappers over system calls & all other normal library functions) and append them to the *function_table*
3. disassemble the code. For every function call instruction:
   - determine the address in the file where we found the instruction
   - determine the caller by finding which function in *function_table* contains in the *[begin address, end address]* interval the current instruction address
   - determine the callee by finding which function in *function_table* has the start address equal to the one to which the code wants to jump to
   - add a caller->callee entry in a function call graph
4. build a tree of functions called beginning with main() (or _start() or _init() or whichever compiler dependent function the user may want to begin the tree from). Mark all redundant functions in the code (functions that are not called directly anywhere in the file and which are or are not marked as static in the source code). Alternatively, we may select a tree of functions that are called with a user specified tree root.
5. output to the user:
   - the tree of functions computed at the last step or
   - a list of functions (or connected components of the graph) that are not part of the main subtree – functions that may not be called directly and which are possibly dead-code.

This seems straightforward but on closer inspection be see some flaws of this implementation:

- we don't consider *signal*(*signum, &sighandler_func*) cases
- we don't consider calls to a function through a pointer (attributing the address of a function to a pointer and then calling the value in the pointer).
- having *signal* itself as a pointer and the *sighandler_func* as either a direct address of a function or as a pointer to a function too.

Solving the first case is rather simple:

- before calling signal the compiler pushes on stack the two parameters.
- we determine these parameters by disassembling instructions preceding the call to signal:

  go upwards in the code and check the last two values to be pushed on the stack. The one before the last represents the address of the function being registered as a signal handler. We look for the pushed value in the *function_table* and mark the function that begins at that address as a signal handler function (this means that, although it's not directly called from within main()'s subtree it may be called by a signal).

Solving the second case, that of functions called as pointer to functions is a bit harder, but nevertheless implementable. We'll find somewhere in the code a call with the function address in a register. We shall have to walk up the function (and maybe the function call tree if necessary) to determine which is the function that may be called through the function pointer. We'll have to look for all the *mov* instructions, but also for *add* and other instructions too. For example a weird compiler may say something like:

```
xor eax, eax
add 0x8048324, eax
call eax
```

where *0x8048324* is the address of a function that is being attributed to a function pointer. After analysing code generated in different situations by common compilers *movs* and *adds* should be enough considering that we're not compiling in release mode and compilers don't usually do unnecessary math on these function pointers.

One other building-block of this approach is the determination of variables. Dwarf info in the elf file provides sufficient information: where the variable is stored (stack of some function, global data, function static variable), which is the initialization value (the value that is in memory at the address of the variable), the type of the variable and the length in bytes. We can monitor access upon a variable by watching which *mov* instructions have that address as a source/destination. We can then enforce that some variables (e.g. global buffers) may not be accessed from both from signal handlers and from process context (if only one of the contexts writes to the variable the use of the value of the variable may become uncertain and special measures must be taken to prevent asynchronous errors).

Defect Models

Upon these elements (function calls, argument passing, variable access) a static binary checker may check for some other common programming errors (this list is by no means exhaustive):

A) Functions called in interrupt context

Sometimes it is a dangerous thing when functions called in interrupt context modify global variables accessed by other functions running in process context. Lets say that a function is testing a global variable as such:

```
if (x != 0)
     y /= x;
```

We have a signal handler that set the *x* variable to 0. Let's suppose that the first function passes the `if (x != 0)` test and right after that gets interrupted and the signal handler is called. When control is given back to the first function a divide by 0 error will be generated and the program will

most probably exit.
The static checker should warn the programmer when such things might happen. We will do this by searching for every signal handler function and by verifying if it modifies global variables that other functions access.
For each *call* instruction from the disassembled code we find the caller and callee with the algorithm presented earlier. If the caller is a the *signal*-family system call, we mark as a signal-context functions the callee and all functions called by it (which we find by parsing the call graph).
The next task is finding all the global variables accessed by the signal-context functions and checking weather they are used also in normal process context and seeing weather any of them are misused (not used within atomic operations). We must also check for stdio calls from within the signal-context because stdio functions like *printf* modify global buffers. One call to *printf* may be interrupted by a signal handler. After that handler calls *printf*, which modifies the buffers and stdio global variables, the control is given back to the first call to *printf*. This one is unaware of the changes that occurred during signal handling and in the best case scenario, it will overwrite some of the output of the signal handler, but it may also cause a segmentation fault and crash the program. This kind of checks must be done not only to the stdio functions but to all all signal-unsafe library calls.

B) using uninitialized pointers

From DWARF information we get every pointer declaration and check every instruction until the end of the current function. We can detect two kinds of programming errors:

- find weather a pointer is used without being first initialized (and raise an warning if one is found)
- find an allocation of that pointer with a *malloc*-family functions, *strdup*, etc. and find that the pointer value is not checked against NULL before being first dereferenced.

C) double *free*

After finding a call to the *free* on pointer in the program, we will search for another call to *free* that is using the same pointer. A warning will be raised if a second call is to be found.

objdump output:

```
sub    $0xc,%esp
pushl  0xfffffffc(%ebp)
call   8048308 <free@plt>
add    $0x10,%esp
sub    $0xc,%esp
pushl  0xfffffffc(%ebp)
call   8048308 <free@plt>
```

D) using a pointer after *free*ing it

After *free*ing a pointer, we will check from there on weather that pointer is ever dereferenced again and issue a warning if this is the case. This is different from the last case because we don't dereference the pointer here.

objdump output:

```
sub    $0xc,%esp
pushl  0xfffffffc(%ebp)          ;free the pointer
call   8048308 <free@plt>
add    $0x10,%esp
mov    0xfffffffc(%ebp),%eax     ;write to the freed pointer
```

E) casting misuse

Example:

```
char c;
int *p = (int*)&c;
*p = value;
```

In some cases, casting from *char** to *int** might be what the programmer intended. But in the case of the previous code snippet, though legal,

```
*p = value;
```

will overwrite stack data which may lead to remote code execution bugs or just plain-old core dumps. DWARF has type information about all variables allocated, and for each type it stores the size in bytes. Checking this kind of casting misuses is therefore a possible task which can be done in parallel with the call graph generator.

F) buffer overflow

This issue is most dangerous with stack allocated buffers because many architectures hold function return addresses on the stack. When a function calls *ret* the program counter register (which holds the next instruction to be called) will be set to the value from the top of the stack. If a buffer overflow occurs, this address might be overwritten and set to a location where some remote code exists, enabling it to run.

Example 1:

```
char a[10];
strncpy(a, ..., 20);
```

Example 2:

```
n = 11;
char a[10];
read(fd, (void*)a, n);
```

We can determine the size of the stack allocated buffers and determine if they are being misused in any particular way. For this we must create a table of all standard library function calls that require buffers and note which are the appropriate calling methods (*strncpy* must have the last argument equal or less than the size of the buffer pointed by the first parameter, etc.). Some parameters might not be

pushed onto the stack but taken from a memory location (as within the second example). This can be easily solved by storing for each variable the value that was given to it (if the value is hard coded in the binary).
A second form of buffer overrun is that of heap overrun. This is not as dangerous as the first but it may cause damage to heap-stored data. For dynamic allocated pointers, we can get the allocated size only for some calls like *malloc(100),* or *malloc(n)* when we can compute directly from the binary the value of *n* (it might be the case of a const, or a just initialized variable).

G) use of uninitialized variables

Using the value of a variable before being initialized.
Example:

```
int test(int k)
{
     int n;
     if (k == 0)
          n = 1;
     return n;
}
```

We will try to catch this kind of errors by verifying all program paths to see if stack allocated variables are read without ever being written.

H) misused sizeof()

Example:

```
TCHAR buff[50];
_tcsncpy(buff, inputstr, sizeof(buff));
```

TCHAR represents either a *char* or a *wchar_t* and *_tcsncpy* is macroed to *strncpy* or *wcsncpy*. A program developed and tested for ANSI strings, ported to use wide chars might have this kind of code. This issue will not be caught until someone creates a 50 wide-char `inputstr` string. This is actually a sub case of **F)** because the complier will replace all *sizeof*(*varname*) with the actual size in bytes of *varname.*

I) misused memset()

Executing a memset call with the number of bytes to be set being zero.
It's a common programming error to have inverted the last parameters of the *memset* functions. Most commonly a programmer uses *memset* to wipe-clean a memory buffer: set it all to zero. Because both of the last parameters are integer values they can be switched by mistake. This will never generate an error, but depending on the fact that that buffer is all zeroed out, sometimes will. And having in the code an instruction such as:

```
memset(s, 20, 0);
```

will only cause harm to a developer that reads the code. He'll think s has 20 zeroed out bytes. In fact, nothing gets done after this call.

objdump output:

```
push   $0x0
push   $0x14
pushl  0xfffffffc(%ebp)
call   80482e8 <memset@plt>
```

We will look for *memset* calls and get the last parameter pushed onto the stack. If it's zero we will raise a warning and present possible solutions: either remove the memset call (to make the code less ambiguous) or swap the two parameters (so that the function zeroes the buffer).

For further developing we should focus on cross function analysis, recursive function calls, return function values and defects discovery optimization. We should also try to develop more defect discovery models by studying the most frequently used library functions on Unix systems (*fopen, open, malloc* ...) and the programming errors that appear most often when using them.
For example after a call to *fdopen*, the file descriptor should not be used anymore until there is an explicit *fflush* call on the *FILE** returned by *fdopen.*
Another example would be the flushing of stdio buffers before a *fork* call.

The application should be written with extensible support in mind, especially for adding new defect model discovery modules. Command line parameters should disable some types of error checking, so that the user can select which defects the application should look for and not waste CPU cycles for defects that programmer is not interested in, or has previously checked for.

Also after the application will have integrated a significant number of modules it should be tested for incorrect warnings. A large number of false results will render it an unusable tool, so reports must be made only if we're sure that the binary contains an error.

Bibliography

DWARF2-XML - http://www.sde.cs.titech.ac.jp/~gondow/dwarf2-xml/
DWARF Debugging Information Format resources - http://reality.sgiweb.org/davea/dwarf.html
DWARF2 debugging information format - http://ratonland.org/?menu=3
Wikipedia - http://en.wikipedia.org
Linkers and Loaders - http://www.iecc.com/linker/
PREfix & PREfast - http://research.microsoft.com/collaboration/university/