1 Introduction

2 Background

2.1 Information Flow Analysis

Information flow analysis tracks interactions of information throughout a program. A simple application of this analysis is used in LOMAC to restrict access to data of various integrity levels. LOMAC maintains a concept of information integrity for each object, and restricts information from flowing from low integrity objects to high integrity objects. If a program source is inspected, and data is given a level of integrity or confidentiality then by following the flow of information in the program, it is possible to track which data have been computed from confidential or high integrity information. In cryptographic systems, the encryption keys are confidential, if other data within the program is calculated from the encryption key, there is a flow of information from the confidential data to the computed data. Compilers also use information flow for optimization purposes. By identifying what information is used and when, alterations to a program can be made to improve cache locality, hide memory latency and improve performance in other manners. For program level security, a compiler could be able to check the level of integrity for any variable through the use of information flow. By tracking confidential or sensitive pieces of information, any variable or decision which is computed from a confidential data has a flow from the confidential data to the variable or branch. An attacker may be able to exploit parts of a program which are dependent on some confidential data, and with sufficient flow, can possibly reconstruct the confidential data. An example is given in Algorithm 1 to illustrate this concern.

Algorithm 1 Simple Information Flow

```
$k$ = sensitive information
$d.x = k + 1$
$d.y = 0$
$a = -1$
**if** $d.y == k/2$ **then**
    $a = 4$
**end if**
```

In this simple example k is confidential data, through information flow we can see that the value of $d.x$ is directly computed from k. The value $d.y$ however is not confidential due to its value having no reliance on the confidential data. The flow between k and $d.x$ is called explicit flow, since the value of $d.x$ is computed from the confidential data. As the analysis continues, any value which is computed from $d.x$ will also be treated as confidential data. There is also implicit flow which can be seen in the first branch on $d.y$ where it is compared with some value computed from k. In this case $d.y$ is still considered non-confidential data, since the value has no relation to k, but there is a flow from k to a indirectly. Although, a is not computed from k, after the branch executes, the value of a may or may not have changed. If it does change, the value of k has been determined. If it does not change, the value of k is not determined, but a possible value has been ruled out. This indirect flow from k to a is what is called implicit flow. Tracking the flow of information between variables can be done by creating a set of constraints for each instruction to represent the direction of flow. Creating these constraints with mutable pointers, requires more information than is provided from IR code from the compilation process.

2.2 Points-to Analysis

A Points-to analysis is a reference tracking analysis to identify the targets of pointers in a program. Information flow alone is not enough once pointers are involved. For example, in Algorithm 1, it must be possible to identify from the source the location of d, and subsequently the location of x. The points-to analysis provides this information so that the proper location of x and y can be known even if the value of d changes, meaning it points to another location in memory. In addition to providing spatial information, the points-to analysis used in this research provides type and data layout information. The analysis accomplishes tracking by maintaining a internal graph representation of memory throughout the program. The graph is made up of nodes which are units of memory that are able to be the target of a pointer. Data structures identified in program source have one node for an instance of the data structure, where all fields pertaining to that data structure are represented by that node. The edges in the graph are references, so it is possible to identify where the target of a pointer resides. Additionally, this analysis provides type information along with offsets of the type from the base pointer. In the baseline tainted flow analysis, the granularity of the results was achieved by creating constraints of information flow between nodes from the points-to graph. This however, as will be discussed, leads to an over-approximation of the resulting tainted values. The points-to analysis used also has a method of determining the validity of the type information.

2.3 Timing Channels in Crypto-systems

A timing channel is a side channel in which an attacker uses execution time to learn information about sensitive data. In some implementations of sliding window exponentiation, the sequences of squares and multiplies could be measured due to differences in each methods execution time. This attack though timing based can be observed in source code in OpenSSL's previous implementation of sliding window exponentiation, shown in Algorithm 2.

Algorithm 2 Square and Multiply Timing Channel

```
for (i = 1; i < bits; i++) {
    if (!BN_sqr(v, v, ctx))
        goto err;
    if (BN_is_bit_set(p, i)) {
        if (!BN_mul(rr, rr, v, ctx))
            goto err;
    }
}
```

The for-loop here iterates through the number of bits in the confidential power `p`, each time through squaring `v` until a set bit is encountered in `p`. Once a set bit is encountered, an additional multiply step is executed before proceeding to the next loop iteration. This lends itself to a timing attack due to multiplication more expensive to compute than a square. The timing channel exists because the number of operations executed during one iteration changes based on the value of the confidential data. The result is if one can determine the locations of the multiplies and the length of time expected between each multiplication, the key can be rebuilt as has been done by Bernstein et al [1]. There are other forms of timing channels which may or may not be viewable in source code, such as disk access timing or network timing attacks.

3 Improving Field Sensitivity in Taint Analysis

In this work, improvements were made to the precision of the baseline taint analysis. The baseline represented each structure or array as one entity, which led to imprecision when tainted data flowed to only some parts of an entity which had multiple fields or elements. Simply, the improved analysis generates multiple elements to sufficiently represent different data structures and constrains the appropriate ones based on the operation. Improvements in precision also increased the accuracy of the analysis by reducing the number of false positives from the final results. The analysis is comprised of following 3 components: Flow capture, constraint generation, and solution.

3.1 Information Flow Capture

The first component of the analysis which identifies the information flows of the program, works on the instruction level. For each instruction a set of flows will be identified, called a Flow Record. Each instruction will generate an explicit and implicit flow record, which yield different results depending on the type of analysis run. The analysis attempts to start from the main function, but if it does not exist all functions will be analyzed as the start point. Any function which calls other functions within its definition, is analyzed in its own context, so information can flow through the arguments between two functions. For each function context, all instructions are parsed to identify operands and operation. The operands are divided into source and sink operands and based on the operation. The flow is added to the corresponding set based on whether or not flow is explicit or implicit. This section has not been modified in the improved version of the analysis.

3.2 Constraint Generation

After the information flow capture component is finished, there are a set of flow records for each function context and this stage will generate the corresponding constraints for each flow record. The changes to improve the baseline analysis are mostly done in this stage. Once the flows through the program are identified, each constraint which makes up the set, is generated by iterating through the flows. A constraint is added to the set constraining the sink element in relation to the source element.

The first time an instruction is encountered as either a source or a sink, a constraint element would be generated. For most instructions, just one constraint element should be generated as the instruction serves as a source or a sink to only one value in the program. For operations which load from compound data type, such as a class, struct, or array, the baseline generated one constraint element per data type instead of one for each field within the type. This lead to imprecision particularly when sensitive data flowed to a part of the compound type, the field it flowed to would be tainted as well as all other field which shared the same memory location. The GetElementPtr (GEP) instructions in LLVM is one instruction in particular that is used when getting a field at some offset from a base address. The improved analysis treats this instruction differently than others to achieve a more precise result.

GEP instructions are the intermediate level instruction used when addressing a field within a compound type. These types are treated as pointer types and all fields are offsets from them, so the points-to analysis is used to create elements unique to each instance of a compound type. Each unique compound type is captured by recognizing the GEP instruction source or sink, and finding the corresponding memory location it addresses with the help of the points-to analysis. Once captured, if the data layout can be determined for the type, a constraint element is generated for each field in the type at the offset assigned by the data layout information stored in the points-to analysis memory node. After this set of elements is created, the field offset is an operand within the instruction is used to pick the corresponding element to be constrained. The rest of the instructions are handled the same as the baseline. After iterating through the flow record, with this change, the set of constraints generated has been changed to be specific enough to have flow between elements within a compound type without tainting any accompanying fields.

3.3 Solving Set of Constraints

Given the set of constraints for the program being analyzed, the smallest solution where all the sinks are at least as large as their sources is found, otherwise known as a least solution. From the solution each sink operand will have a value is greater than or equal to that of the initial confidential data, then that value is identified as being confidential as well. In the case of crypto-systems, it can be helpful to find all the values which may have been computed from confidential data. These values are possible locations for leakage of confidential information, and an attacker may be able to exploit the computed confidential data to learn information or reconstruct the original confidential data.

Consider Algorithm 1 for example of how the analysis functions. First the flow of information must be identified. For simplicity, this example will just consider the source code, not the instructions. The more detailed analysis will be done in the implementation section. In Algorithm 1, k is just a constant set to some confidential value, no information flows there. Next, $d.x$ is calculated by adding 1 to the value of k, so there is flow from k to $d.x$. With $d.y$ and a, in lines 3 and 4 respectively, they are set with constants, so no flow happens there. Then, a branch is encountered where the value of a is modified based on the values of $d.y$ and k. Let the branch condition be b, there will be flow from $d.y$ and k to b. Lastly due to the branch, there is an implicit flow from $d.y$ to a and k to a.

For explicit flow the set of constraints is as follows.

$$k \leq d.x$$

$$d.y \leq b$$
$$k \leq b$$

In this case it is easy to see that $d.x$ should be at least as confidential as k since it is directly computed from k. Similarly, the branch condition b is confidential since it must be at least as confidential as k. The value $d.y$ is not confidential, but is irrelevant because confidential information is used to determine the outcome of the branch.

If implicit constraints are to be considered then the set is as follows.

$$k \leq d.x$$

$$d.y \leq b$$

$$k \leq b$$

$$b \leq a$$

In this case the all the constraints are the same apart from the addition of the last constraint. There is another constraint between the branch condition and a, because after the branch, the value of a is dependent on the outcome of the branch. If an attacker had a method of inspecting the value of a, they would learn information about b which is derived from k.

The baseline analysis generates a set of constraints similar to the examples above, but with some imprecision. During the stage of generating constraints from the information flow, the points-to analysis is leveraged to figure out what is being pointed to by the variable d to subsequently find the correct instance of x. The points-to analysis will represent the data structure pointed to by d with one node in the graph, and the constraint generated refers only to that node, instead of to the specific field in the node. The effect on the constraints is as follows for explicit flow.

$$k \leq d$$

$$d \leq b$$

$$k \leq b$$

The result is that $d.x$ and $d.y$ are indistinguishable from each other and the results will report both $d.x$ and $d.y$ as confidential even though manual scanning of the source shows that $d.y$ is never calculated from confidential data. This is a common problem when analyzing sources. Take this implementation of AES, where the data structure used is comprised of both confidential and public data. The AES_KEY struct has the confidential rd_key and a second field rounds that is considered to be public. Algorithm 3 shows where the inaccuracy can be found in actual source code.

So with the improved analysis, the constraints are generated such that compound data structures with multiple nodes are able to be constrained separately. Once the set of constraints for a programs source code is created, the least solution is found and the results show the line number where the branch occurs. This list of results shows the branches which are tainted and untrusted.

Algorithm 3 Public and private data in structure

```
struct AES_KEY_t {
  unsigned * rd_key;
  int rounds;
};

typedef AES_KEY_t AES_KEY;

void AES_enc(char * in, char * out, AES_KEY) {
  unsigned * rk = key->rd_key;
  if (key->rounds > 10) {
    ...;
  }
}
```

3.4 Result Classification

The analysis provides a list of source code lines which are vulnerable. Based on the contents of the input files, a branch is flagged as vulnerable if the condition depends on data which has been marked as tainted or untrusted. The reported lines can then be reviewed and sorted to rank them in order of severity. Classification of the vulnerable branches consists of three stages, the first removes error-handling results, the second sorts the remaining results into either a high or low-risk set, and the final sorts the high-risk results depending on the surrounding source context.

Stage 1 is a filter which removes the error handling and input validation results. For normal operation of a program, these branches check sensitive data but allow an early exit if the branch condition evaluates to true. In Algorithm 4, the variable `N` is tainted, so the branch is vulnerable due to the condition depending on `N` and the data held within `N`. The result of this branch is not helpful in the case of an attacker attempting to learn the value of `N`. As long as the input is valid, the condition of this branch will always evaluate to false. If the branch leads to exit code, it is a candidate for removal in this stage.

Algorithm 4 Validation Source Code

```
if( mbedtls_mpi_cmp_int( N, 0 ) <= 0 || ( N->p[0] & 1
    ) == 0 )
  return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );
```

Stage 2 operates on the remaining results to sort them into high and low-risk categories. A high-risk branch will be due to one of the operands being either directly related to the sensitive data or derived from more than 1 bit of sensitive data. Low-risk branches, are branches which the operand is based on 1 bit of the key or the same operand could derived from some set of sensitive data values.

Algorithm 5 High-Risk and Low-Risk Branches

```
1  #define MPN_NORMALIZE(d, n) \
2  do {                        \
3    while( (n) > 0 ) {        \
4      if( (d)[(n)-1] )        \
5        break;                \
6      (n)--;                  \
7    }                         \
8  } while(0)
9
10 MPN_NORMALIZE(ep, esize);
11  if (esize * BITS_PER_MPI_LIMB > 512)
12   W = 5;
```

Algorithm 5 is a snippet of some of the modular exponentiation code from Libgcrypt 1.8.2. For instance, say the values pointed to by `ep` are marked as tainted at the start of the analysis, and the analysis reports that lines 10 and 11 are vulnerable. Line 10 is the macro defined in the lines 1-8. Line 10 is reported due to the macro having the `if` branch directly dependent on the data pointed to by `ep`. Since the branch is directly dependent on the tainted data that branch is considered high-risk. Line 11 however is a branch based on `esize`, the value begin modified within the `MPN_NORMALIZE` macro. The value of `esize` is not unique to the tainted data, meaning there exists a set of tainted values which may yield the same `esize`. If a value is derived from tainted data, as `esize` is, but the result is not unique to that instance of tainted data, it is considered low-risk.

If for any reason, the classification is unclear, such as a mutable type being passed to source code that is not analyzed, the data is treated as high risk. This is done in order to be conservative. Further categorization can be performed on the high-risk set of results in Stage 3. Additionally, if there are a large number of low-risk result, Stage 2 can be repeated, by appending the low-risk variables to the whitelist.

Stage 3 specifically looks a the context of the high-risk results from Stage 2. Each of the results will be classified as either a single branch, repeated branch, controllable branch or in extreme cases both controllable and repeated. Single branches are branches which are not within a loop. Repeated branches are branches that are part of a loop definition or within the body of a loop. Controllable branches are that which the branch result is both untrusted and tainted. The purpose of sorting results this way is to be able to prioritize which vulnerable branches to examine first when attempting to make the application secure.

4 Evaluation

4.1 Implementation

Achieving the improved precision of the analysis, is done primarily through handling LLVM's pointer instruction differently than other instructions. The pointer instruction in the IR is the GetElementPtr instruction. This instruction is used when computing an address from a base pointer. For arrays, for example it has the index of the element and the bounds of the array if they are known. For structures, the instruction contains the structure type which is being addressed and the index of the field in the structure that is being referenced. The baseline analysis lacked the ability to consider the index available in these instructions. The points-to analysis had type and offset information for the targets of the pointers. Improving the analysis was achieved by i) creating the appropriate number of constraint elements for each node in the points-to graph, ii) converting the field index from the GEP instruction to a byte offset, and iii) constraining the correct elements based on the GEP instruction and data type information.

4.1.1 Constraint Element Generation

When a pointer value is encountered for the first time, a node is created for it in the points-to graph, and a set of constraint elements are created for that data structure. The points-to analysis was left unmodified, so the focus will be on the semantics of how constraint elements were changed. In the baseline, only one constraint element represented any target of a pointer. In the improved analysis, type information from the points-to analysis was used to generate the appropriate number of constraint elements. In a stack allocated array, one constraint element was created for each element in the array. For structures and classes, each field is located at some offset away from the base address of the structure and a constraint element was created for each field and mapped to associated byte offset. This change enables the precision of the baseline to be improved by correctly selecting the corresponding constraint element based on the IR.

For any GEP instruction, constraint elements are generated using the type information from the points-to analysis when available. Each constraint element is created by iterating through the type information of the node from the points-to analysis. For each type there is an associated byte offset, using this offset and the size as reported by LLVM, a constraint element is created with the corresponding starting byte offset and the ending byte offset based on the width of the field. It is necessary to account for the width and start location of each field because there may be padding between fields of a structure. Padding between fields may be present in the type information because of an unused field in code. The points-to analysis will not have type information on unused fields so it is important to account for these gaps. Another reason to account for the padding is due to structures which are not aligned with each field following sequentially after the previous field. When the type information is not available one constraint element is created for the whole node so at worst the improved analysis will be equivalent to that of the baseline.

Algorithm 6 shows how the points-to analysis is leveraged to create constraint elements for the proper byte offsets and widths. Let `s` be the LLVM representation of the structure type referenced by the GEP instruction, and `node` be the node in the memory graph provided by the points-to analysis. The algorithm works by retrieving the graph in which the node resides and then retrieving the data layout as specified by the compiler for that graph. The data layout contains information like the field lengths of each type and the alignment within a data structure. The structure type itself `s` does not have the alignment and padding information, only the types of each field within the data structure. The data layout is used to retrieve the structure layout of the structure type from the GEP instruction. The structure layout is vital because it allows the index from the GEP instruction to be converted to a byte offset within the data structure. The data layout also provides the size of each type, so that the constraint element is created at the corrects starting and ending offset.

Algorithm 6 Creating constraint elements for each field in a type

```
const DataLayout &TD = node.getParentGraph()->
    getDataLayout();
const StructLayout *SL = TD.getStructLayout(s);
int index = 0;
for(Type::subtype_iterator it = s->element_begin(); it
    != s->element_end(); ++it, ++index){
    unsigned start = SL->getElementOffset(index);
    unsigned end = start + TD.getTypeStoreSize(*it);
    std::string label = "[" + std::to_string(start) +
        "," + std::to_string(end) + "]";
    const ConsElem & elem = kit->newVar(name+label);
    locConstraintMap[&loc].insert(std::make_pair(start
        , &elem));
}
```

The elements created using this strategy enable the analysis to be more precise. GEP instructions can also be used to index arrays, so that is handled by computing the number of elements that array may hold. For stack arrays that number is known, but for heap arrays the number is variable. For stack arrays each, since the number of elements is known, one constraint element is created per array element. For heap arrays, one constraint element is created for the entire array.

Similarly, if type information is unavailable then a conservative approach is used. One constraint element is created for the entire data structure. When the information is available, the analysis is able to be improved by constraining the correct constraint element for each instruction. When that information is unavailable, the improved analysis is unable to be more precise than the baseline analysis.

4.1.2 Constraining Operands

Each time an instruction where information flow exists, constraints are generated between the operands of that instruction. The idea of this is simple, given a set of constraint elements select the ones which are used in the operation and constrain them as necessary. The baseline analysis handled individual variables in a precise manner as long as the variable was not a structure or array. The baseline analysis had no way of identifying the offset from a GEP instruction, and even if it did, the mapping between a memory node to a constraint element was one-to-one. In the previous section, the mapping from memory node to constraint element was made to be one-to-many. As one node may represent multiple fields which together make the structure or array. It is now necessary to pick the correct element from this one-to-many mapping when generating the constraint for each instruction. Again, this is done by analyzing the GEP instruction.

The last operand of a GEP instruction is the index of the field within the data structure. Each constraint element for a memory node is created for a byte offset range, so it is necessary to correctly calculate the byte offset from the field index provided by the GEP instruction. This is very similar to the strategy used to generate the constraints at the correct byte locations. Given the structure type, field index and node from the points-to analysis, the offset is calculated as shown in Algorithm 7. Given the node, the data layout and associated structure layout can be found and then the index can be converted to a byte offset.

If the GEP instruction is in reference to an array access, then the offset is just calculated using the size of the element type and the index. If no type information exists for the node, only a single constraint element would exist for the node, so that is the constraint which is selected to be constrained.

Algorithm 7 Calculating byte offset from index

```
unsigned findOffsetFromFieldIndex(StructType* type,
    unsigned fieldIdx, AbstractLoc* loc) {
  DataLayout &TD = loc->getParentGraph()->
      getDataLayout();
  StructLayout *SL = TD.getStructLayout(type);
  return SL->getElementOffset(fieldIdx);
}
```

4.2 Benchmarks

4.3 Case Study

5 Related Works

6 Conclusion

References

[1] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.