# MemSan: Progress report

Hugo Genesse
*Computer Engineering*
*Polytechnique Montreal*
*Montreal, Quebec*
*hugo.genesse@polymtl.ca*

***Abstract*—MemSan is a user-supplied data flow analyzer targeting memory-sensitize functions like strcpy that could lead to buffer overflow vulnerabilities.**

## 1. Introduction

Data flow can be hard to follow for programmers. User-supplied data can then be used without proper checks with standard functions that can have bad repercussions on the security of the product they are developping. On the other hand, security researchers spend time reconstructing that flow in their head to find exploitable paths. Our goal is then to facilitate the reconstruction of user input to certain function calls that would be interesting security-wise. MemSan uses the clang compiler frontend to recreate an AST that will be analyzed to identify problematic coding patterns.

## 2. Construction of MemSan's AST

### 2.1. MemSan's custom AST format

The first step is the reconstruct a simplified AST that will be processed later. We produce our own AST representation because our tool is aimed at specific goals and would be obstructed by the number of attributes the original clang AST. We are then decoupled from clang and can more easily implement new parsers and use the same code for the same analysis. For example, our AST output for the same code fits in 20 lines of XML instead of 81 lines of a the custom clang AST output format with also has longer lines. As you can also see, we have full control of our format to ease our post processing.

### 2.2. Implementation

Using clang's RecursiveASTVisitor, some methods were overriden to redefine the behaviour the visitor on specific elements that we wanted to analyze. We then redefine 5 different traverses and 2 visits. The redefined traverses are the following: TraverseWhileStmt, TraverseIfStmt, TraverseCXXMethodDecl, TraverseCXXRecordDecl, TraverseFunctionDecl. We chose to override traverses for those because they can encapsulate other elements we are interested in and it is then important to visit its children. For example, a class will have methods definitions inside of it and we need information on the latter to complete our analysis. Because the other 2 elements, VarDecl and BreakStmt, are simpler elements that can't encapsulate others that we want to analyze, visits sufficed to create opening and closing XML tags for our custom AST. The XML format was chosen for its wide use and tree-like structure that fits ASTs well.

## 3. Code Metrics Extraction

### 3.1. Implementation

The code metrics extraction is written in Python3 and entirely decoupled from clang. It resides in the ASTAnalyzer class and leverages the xml module from Python3's standard libraries to do its job. It starts and finishes with a 'dump' tag that encapsulates each files that were processed. Afterwards, we look for every 'filename' tags to identify files, then classes, methods and so on. The information we gathered is then written in a CSV file with the following format: ID, FILENAME, CLASSNAME, METHODNAME, #IF, #WHILE, #BREAK, #VARS.

## 4. Conclusion

We succesfully implemented our own custom AST format in an XML file. Code metrics on methods were then calculated externally in Python to make our analysis decoupled with clang. The custom AST also simplified the original AST to make our analysis simpler and more focused on our needs. A CSV is then produced to visualize the code metrics we calculated.

## Acknowledgments

Nicolas Cloutier, best teaching assistant.