Reverse Engineering and Exploit Writing

TA-2 Assignment

Guided By

Mrs Meena Lakshmi

(Assistant Professor)

Submitted By

Raj Shekhar

Enrolment no.: 240545002004

M.Sc. in Cyber Security

Semester I

School of Cyber Security & Digital Forensics

National Forensic Science University

Bhopal Campus, M.P., 462001, India

Day 1 - What is RE and What Can it Tell?

Reverse Engineering (RE) is clearly defined when it is about to deep dive into the internal workings of software, uncovering its hidden structures and logic within assembly code. This knowledge can be particularly useful for deciphering cryptographic algorithms, diagnosing system crashes, or even exploiting vulnerabilities in input parsing mechanisms.

Day 1, Part II

In this lesson of how negative numbers are represented using two's complement is solidified, through the process of flipping bits and adding one. In terms of assembly logic, the operations CMP (compare) function as a subtraction process, while TEST operates like an AND operation without storing the result. It is important to note that these instructions do not store results but instead set CPU flags for subsequent conditional jumps. As for local variables in functions, they are generally referenced from the stack relative to the Base Pointer (EBP). Similarly, array accesses utilize a logical approach involving Base + Index * Scale + Offset. Finally, it was elucidated that IDA employs recursive traversal to decode instructions rather than sequentially reading bytes.

Day 1, Part III - Knowing about the Tools

The understanding of IDA's role as the central tool to be utilized was revised. It highlights its recursive traversal disassembly feature, which facilitates a deeper level of analysis. The key shortcuts such as G to navigate directly to specific addresses and the Escape key to return to previous locations were also highlighted. The interface provides valuable insights with color-coded lines for conditional jumps, indicating when a jump is taken or not. Then, additional tools supporting IDA were introduced, including PEid for quickly identifying compilers and crypto tables using plugins, and CFF Explorer for parsing the PE header and viewing imported libraries/functions.

Day 1, Part IV - Basic Reverse Engg. Algorithm

Reverse engineering is structured around simple iterative steps that are essential for success. Initially, it involves gathering initial insights by studying strings and imports, which serve as the first step in understanding the program's behavior. Dynamic analysis follows, where running the program provides crucial data to identify specific functions of interest.

Then next phase focuses on analyzing these selected functions in depth to reveal underlying algorithms and data structures within their codebase. This recursive process is repeated until a comprehensive understanding of the program's workings is achieved, making this structure particularly effective for complex binary programs.

Day 1, Part V - Applying the Algorithm to the Bomb Lab

Following the video which guides a practical demonstration of how the reverse engineering algorithm works in practice, by following it step-by-step process using a program named "bomb". It is been found that the core strategy was to exploit the readily available information from the

printed string, that is : "Welcome to my fish little bomb" prompt and use it for an immediate insight into the program's functionality.

Now, emphasize in utilizing the IDA strings window to locate this specific string within the program's data structures. This allowed to pinpoint a crucial piece of code that provided an entry point rather than beginning from scratch without any context. The cross-reference (X key) function in IDA was then used to find where the prompt is called within the main function, in the first phase.

Day 1, Part VI - Lab Summary

The Phase I walkthrough was pivotal in understanding how reverse engineering can be streamlined, particularly when working with programs that generate similar strings upon explosion. It highlighted that tracing backwards from the immediate aftermath of an event (in this case, the explosion function) is less efficient than tracking a string like "Welcome."

In the main codebase, the input validator function was identified as a key structure due to its frequent invocation before each phase check. This renaming of the function and other critical structures with the N key proved beneficial for improving readability within the assembly code.

Following the video which also demonstrated how analyzing the internal logic of call functions could lead to recognizing common assembly patterns such as those used for calculating string length and performing a string comparison, which in this case, helped to identify the Phase I password.

Day 1 Part VII - Creating the Lab Answers File

The tip regarding the use of command line arguments to save time on repetitive tasks became quite handy in practice, and can be used for automation that could potentially, reduce redundancy in data entries. Then, the spacebar key for toggling, if one wants to switch between two views of the disassembly window : graphical block view and sequential instruction list view use it. This can be particularly a useful optimisation for viewing convenience or when comparing different code sections side by side.

Day 1, Part VIII - Introduction to New Phase

The key objective for Phase II is to focus on control flow structures such as loops and arrays. It was worth noting that within the loop constructs, there were distinctive patterns that had emerged such as: initialization (setting a local variable), test condition (a conditional jump following CMP/TEST), and the counter or increment section.

The understanding of array access in assembly highlighted that to manipulate arrays, one must understand that they are accessed using base plus index * scale plus offset. The 'scale', it represents the size of the data type in bytes (e.g., 4 for integers). This information provided in the lecture was crucial for interpreting offsets accurately within the assembly code.

Day 1, Part IX - Phase II Walkthrough

In this phase provided an in-depth look at the code's structure and functionality, with a particular emphasis on analyzing input and loops. The function preamble was recognized as being instrumental in setting up the stack frame and allocating memory for local variables. It was emphasized that this setup is crucial because it creates space for data structures to be stored temporarily or permanently.

The initial check uses `sscanf` that confirmed the string input, that had been parsed correctly into exactly six integers, it is suggesting a fundamental parsing of numerical data. The central loop's logic shows subsequent numbers were calculated based on the previous one. This means that the input sequence must have followed a factorial pattern: 1, 2, 6, 24, 120, and 720. This analysis essentially carried out to correctly identify the program's intended behavior.

Day 1, Part X - Phase III Introduction

The key objective for Phase III was to delve into the switch statement, which can be implemented in two distinct forms: a simple series of cascading `if-else if` checks and a more sophisticated technique known as the jump table approach.

In the context of program analysis, it is used to recognized that while a single long chain of `if-else if` conditions could suffice as basic switch statements, but there exists an alternative method which offers significant performance advantages by avoiding direct branching through multiple `if` statements that is using a jump table array to point to various case code blocks.

A valuable advice regarding the scope of analysis in Reverse Engineering was mentioned in the video about how important it is to remain focused on the main puzzle at hand and avoid getting lost down rabbit holes of side-tracking functions that do not contribute directly to understanding the core functionality or logic of the program. And this strategy helps maintain efficiency and focus when dissecting complex binaries.

Day 1, Part XI - Walkthrough

Here the demonstration of the jump table implementation was in practice, providing a look into how the program manages its control flow within a more structured approach compared to a series of `if-else if` statements. The function was found to be designed to handle three inputs: A number, A character, and Another number.

Upon analysis of the first input against a maximum value (7), the program then multiplied that result by 4 (the size of an address pointer) to determine the offset into the jump table array. This method suggests that the program has efficient memory management with each case block having its own dedicated storage location within the jump table, facilitating fast access to the correct code logic without a linear traversal through nested `if` statements.

The use of a number such as '3' allowed for tracing the execution path by observing how each input was processed and compared against specific values or conditions. Following this step-by-

step approach helped in validating the expected pattern needed for Phase IV, and an indication of successful analysis of the control flow mechanism within the program.

Day 1, Part XII - Introduction to Phase IV

This lecture is the introduction to recursion, highlighting the significance of indefinite iteration, which mean functions that repeat their execution until they reach a defined end point. This approach is particularly useful when dealing with repetitive actions or mathematical formulas.

When analyzing recursive functions in assembly, the most crucial step involves identifying the base case, which is also known as the terminal condition. The base case serves as an essential stopping mechanism for recursion and allows the program to terminate its loop of calls without continuing infinitely. It should be noted that any recursive function must have a defined base case so that it can "fold back" upon itself to return a meaningful result rather than running into an endless cascade of self-references.

The writing out of the base case and formula in pseudo-code or table format is considered a powerful strategy for quickly understanding what the algorithm is doing. This technique helps in visualizing the recursive structure clearly and can lead to efficient problem solving within the context of reverse engineering at all stages.

Day 1, Part XIII - Walkthrough

In this phase IV, the video focused on understanding recursive functions with an initial integer input. It became apparent that the core logic behind this function revolved around two critical points: identifying the base case and determining the recursive rule.

The base case for this particular recursion was defined as being a simple threshold value where if the input number was less than or equal to 1, it would return the fixed value of 1. Otherwise, the function would recursively call itself with modified inputs. The difference between these calls was the use of (Input - 1) and (Input - 2) which represented two consecutive numbers in a sequence that is typically known as the Fibonacci Sequence.

This recursive approach to generating a Fibonacci-like number led to an important takeaway: once the base case can be identified and also the formula, solving for the required outcome becomes much more straightforward. The video's walkthrough of this phase emphasized the power of these techniques in reverse engineering by allowing for quick analysis and solution-finding.

Day 1, Part XIV - Debugging

The previous section highlighted points like breakpoints and their classification as software or hardware-based which are important for dynamic analysis in reverse engineering. It was noted that hardware breakpoints are preferred for analyzing self-modifying malware due to their safety and effectiveness compared to software breakpoints.

There are two types of breakpoints: software and hardware. Software breakpoints are often inserted through an opcode such as INT 3, while hardware breakpoints do not modify the code but

rely on CPU registers for halting execution. This distinction is important in reverse engineering situations where a malware or packer might alter its own code during runtime.

Then, the step over (F8) feature allows one to move instruction pointers without executing current function content, while step into (F7) takes the debugger into the specific function's execution. The visual predictions provided by IDA make these steps intuitive and clear for programmers or analysts familiar with code structure.

Finally, the use of command line arguments through the "Process Options" menu in IDA was the highlighted as a method to feed additional information to the debugger during runtime analysis. This feature allows for dynamic interaction between the debugger and the running program, enhancing the ability to analyze behaviors that might not be immediately visible at compile time.

Day 2, Part I - Phase V Introduction

In this, the learning material focused on string decoding and provided an introduction to bit mask operators used in malware analysis. It was emphasized how these techniques are particularly prevalent in reverse engineering, especially when dealing with obfuscated strings within a loop.

One key aspect discussed was the use of AND instruction with the value 0Fh (which is hexadecimal equivalent for the binary number 1001), which zeroes out all but the lowest four bits in a register, effectively masking out higher-order information from character data. The technique is commonly employed to focus on only the lower nibble (rightmost 4 bits) of each character during decoding operations.

Then it was also pointed out that using the AL register for single-byte operations can be a convenient way to indicate to the program to work with bytes only and not larger data structures, such as words or double words. The technique is useful because it allows the compiler to optimize code involving byte access more efficiently.

Overall, this phase highlighted how understanding bit masking and its applications within assembly can be crucial for interpreting and analyzing the behavior of malware that relies on obfuscation techniques to conceal its functionality.

Day 2, Part II - Walkthrough

Further into Phase V, the learning material which focused on string decoding and providing insights into how malware might obfuscate its functionality using bit masking techniques. It confirmed that the input to the program must be a string with exactly six characters. The walkthrough involved observing the execution of a loop that iterates over each character in this string sequentially, using pointer arithmetic to traverse through the data structure.

A notable observation during decoding was the application of an AND instruction with the value 0Fh (which is hexadecimal equivalent for the binary number 1001) on the EAX register. This operation effectively masks out all but the lowest four bits in a character's representation, which means it only processes the lower nibble of each input character.

The resulting low-order nibble from this bit mask was then used as an index to access a predefined array. This design is interesting because it creates multiple correct outputs for a given set of inputs

due to the shared use of the same low-order nibble across different characters in the string. This characteristic can lead to more than one possible decryption, which might be relevant from an analysis perspective. In total, practical examples on how malware obfuscation works by utilizing simple bit masking techniques for a substitution cipher effect is learned. The demonstration of this technique through pointer arithmetic and XOR operations highlighted the importance of understanding such low-level coding in reverse engineering.

Day 2, Part III - Introduction to Phase VI

Phase VI of the learning material focused on graph in a complex context due to nested loops within another loop structure. The strategy introduced was about simplifying large graphs by strategically grouping nodes together. This method involves first identifying and consolidating the innermost loop as a starting point, which is then used to group other loops outside it.

The key takeaway from this phase was that one should not attempt to read through the entire graph all at once but instead work on systematically grouping the nodes into clusters. The order of grouping follows: consolidate the outermost loops first and then move inwards towards the innermost loop's nested structure. Once consolidated, the resulting flattened graph is much easier to analyze.

In practical terms, this approach involves a hierarchical breakdown where each outer loop group is condensed into its own inner set of nodes. This process can be visualized as building up from smaller structures to larger ones until the overall complexity has been reduced significantly. The instructor also emphasized the importance of keeping track of how the groups are formed and collapsed, especially when dealing with nested loops.

A warning point is “No Undo” or “CTRL + Z” functionality in IDA, so it's crucial to save the database at regular intervals during analysis. In the event a group needs to be restructured or if one simply wants to go back and see previous work, closing IDA without saving could lead to data loss, therefore, making sure to save the database of progress through the analysis process, and avoid overwriting saved files to prevent any accidental data loss.

Day 2, Part IV – Introduction II

As Phase VI of the learning material focused on graph simplification techniques commonly encountered in complex data structures, such as those with nested loops. This walkthrough included a practical simplification of these types of graphs by systematically grouping nodes before delving into detailed analysis.

The recommended approach involves starting with the innermost loop and then working outwards to encompass larger groups. This step helps to collapse a large, messy graph into a much more manageable structure for further examination. Upon completion of the analysis phase, it is suggested that one should reverse this process by analyzing from the outermost loops first.

A critical point was the absence of an "undo" feature. To mitigate any potential issues caused by grouping nodes incorrectly, the advice is to save the database as a last resort before closing and reloading IDA. This precaution is crucial when working with large analysis tasks where changes can have significant consequences.

In summary, it was the importance of structured graph simplification for efficient analysis. To ensures the reverse analysis thoroughness and accuracy in understanding the data structure's logic.

Day 2, Part V - Walkthrough

The application of the grouping strategy to the Phase VI graph was witnessed, with nodes systematically grouped inwards from their innermost loop. Before analyzing the logic within the loops, a practical understanding was gained through studying the initial few instructions that revealed the structure as a linked list. By utilizing a custom-defined data structure, called my_items, which included several 4-byte fields and a final pointer field (offset) to the next item, IDA's memory blocks now clearly displayed the chain of elements in the linked list. This approach serves as an efficient means to build context prior to delving into the intricate control flow logic within each loop iteration.

Day 2, Part VI - Walkthrough II

The central analysis of Phase VI focused on the sole purpose behind all those nested loops. Where the initial two loops (Loops 3, 4) were identified as essential for ensuring that the input consists of six unique digits where each number falls within the range from 1 to 6.

Subsequently, the next section (Loops 3, 4, 5) marked a significant stride in understanding, it employs the provided digits (N) to effectively traverse the linked list by moving the pointer N times down the list, followed by an intricate reordering process based on the sequence of input elements.

The final loop (Loop 6) was the pivot, which iterated through the newly reordered linked list and yielded a boom if the current item's field_zero value is less than that of the subsequent item. This behavior signifies that the entire linked list must be sorted in descending order based on the structure's field_zero member.

Recognizing that the input sequence dictates the ordering, and that the final check rigorously enforces a descending sequence, to address this phase effectively.

Day 2, Part VII - Analyzing C++

In this introductory section, the lecture delved into how C++ object-oriented (OO) code translates into assembly instructions, often perceived as intricate but governed by underlying rules.

A key takeaway was the significance of the `*this*` pointer in C++ objects, which serves as a reference to the current instance within an object's context. For Visual C++, this pointer is passed directly into the function using the ECX register. Conversely, for simple classes, the object appears identical to a standard C struct.

When complex class structures are involved, they introduce additional elements that may appear perplexing but follow predictable patterns. Virtual functions and inheritance in particular can lead to a virtual function table (vftable), which is an array of function pointers stored as the first member of the object's structure. These calls to virtual methods then become indirect through offsets within this table, offering a clear indicator of C++ OO code behavior at the assembly level.

Day 1, Part I - (Static Analysis and IDA Pro Basics)

The initial session on static reverse engineering with IDA Pro was comprehensive, emphasizing the significance of having clear targets for analysis prior to delving into the intricacies of assembly disassembly. This approach ensures focused exploration rather than being overwhelmed by a vast array of data.

A fundamental step in this process involves identifying specific program elements such as output strings, function calls, or behaviors that are known. These targeted insights provide direction and context for subsequent analysis.

Key features of IDA Pro were demonstrated during the session. The loading of `bomb.exe` binary was one such example. Additionally, setting up stack variables and exploring view options in the graph view were also covered to enhance visual understanding of program structures.

Then, navigation shortcuts were introduced as well, including `G` for immediate jumps to addresses or functions, which is a valuable tool for efficient traversal through large datasets. The distinction between user-defined functions and dynamically linked library calls was highlighted by their distinct color coding in IDA Pro; dynamic calls are displayed in pink.

Each of these techniques serves to deepen the understanding of data flow and program behavior within IDA's framework. Understanding these elements is essential for a thorough analysis and can be applied to various aspects of software development and security assessment.

Day 1, Part II – (IDA Workflow, Backtracking, and Solving Bomb Lab in Phase I)

The session began with the introduction of good IDA Pro workflow practices, which included saving the database in a `.idb` file to preserve progress. Naming addresses and functions with the `N` key was also emphasized as a method for later reference.

Following this, running `bomb.exe` allowed observation of initial output, which served as a practical exercise. The use of cross-references via the `X` key on output strings like "the bomb has blown up" proved valuable in finding important code flows.

The session also touched upon understanding how array indexing and counters determine active stages within a program. This aspect was highlighted by watching out for the multiplication of a counter by an element's size, such as with a string buffer—80 bytes in this case.

Applying these concepts, the Phase I reverse engineering was completed. By identifying its needed comparison string "public speaking is very easy," it demonstrated how quickly assumptions can be validated against code. This process exemplifies IDA Pro's effectiveness for quick and insightful analysis of software structures.

Day 1, Part III – (Loop Analysis in Phase II & Debugger)

The session centered on Phase II, demanding a thorough analysis of an intricate loop involving arithmetic operations applied to an array of six integers. Key technical takeaways from this phase included :

- Understanding how assembly language handles indexing, utilizing offsets and multiplication (e.g., multiplying by four for each byte).

- Recognizing the iterative mathematical rule governing the generation of each element in the array based on its index and the value of the previous element. This is crucial for understanding the algorithm's logic.
- A brief introduction to debugging techniques: setting software breakpoints (F2) and managing exception dialogs in IDA Pro's debugger.

The session highlighted that while debugging can provide a great deal of insight into memory changes, a deep understanding of the underlying assembly algorithm was more efficient for deriving the mathematical pattern here. This approach proved faster for determining the correct input sequence, reinforcing the importance of both approaches in a reverse engineering scenario.

Day 1, Part IV - (Switch-Case and Recursion Analysis in Phases III & IV)

The project provided valuable experience in handling complex control flow. Phase III focused on an 'if-else if' structure, which uses a single block of code to branch to many destinations based on an input value. This was a significant learning opportunity as identifying this structure with its unique branching behavior became crucial.
Technical insights from Phase III include:

- Recognition of the `switch case` statement and understanding that it branches to multiple outcomes based on a single input value.
- Learning how to manually correct IDA's interpretation of stack variables using Control K and the 'D' key when mixed data types are read by `scanf`. This highlights the need for detailed understanding of memory management in assembly programs.

Moving forward to Phase IV, recursion was identified instantly as a common structure in the code. The ability to translate this into pseudo-code enabled identification of the underlying recursive Fibonacci sequence used for input validation. From here, it was possible to derive the required input without needing to trace the deep recursion. This approach underlines the value of recognizing basic structural patterns and translating them into higher-level representations, which can be easier to work with.

Day 1, Part V – (Decoding Strings, Bitmasks, and Organizing Complex Graphs of Phase V & VI Preparation)

This section focused on mastering techniques for deciphering hidden data, particularly concerning string decoding and bitmask usage, which are prevalent elements found in real-world binaries.
Phase V marked a critical moment as it revealed that a simple bitwise AND operation with the hexadecimal value '0xF' simplifies input criteria by focusing only on the low nibble of each character, reducing the complexity considerably. This insight enabled the identification and manipulation of target letters based on their corresponding index within an array provided in byte form.
In anticipation of Phase VI, a systematic approach was initiated to organize IDA Pro's graph view by coloring nodes or loops based on logical relationships. This strategy aimed to reduce visual clutter and isolate functional units for easier comprehension. Additionally, the session covered the mechanics of defining data structures within IDA to enhance the meaningfulness of offset references (such as 'EAX + 8'), which become more meaningful when dealing with complex data types.

Day 2, Part I - (Advanced Graph Organization and Linked List in Phase VI)

The session involved a significant investment in time and effort to hone the skill of advanced graph organization within IDA Pro's interface. This was essential for Phase VI, which required the ability to dissect nested loops effectively.

Phase VI revealed that the six integer inputs must be unique numbers between 1 and 6, necessitating a deep understanding of the outer loop constraints. The video demonstrated how to recognize linked list traversal in assembly by loading the base address of the current node and accessing an offset (like '+8'), which corresponds to the next pointer value used to load the subsequent node.

In preparation for Phase VI, it was necessary to define the entire structure of a linked list within the data section. This was accomplished through undefining existing memory declarations via the U key in order to create space for a new custom 12-byte structure definition. By doing this, one could easily see and manipulate the numeric values contained within the linked list with ease.

Day 2, Part II – (Understanding Techniques to Resolve Windows API)

The session provided an in-depth examination of Windows API resolution techniques. It highlighted three primary methods for applications to access system functions like 'CreateFile'.

1. Dynamic API Calls: Applications can leverage dynamic loading mechanisms by relying on the loader present in the import table. This eliminates the need for manual runtime intervention, simplifying the process.

2. Manual Resolution at Runtime: For scenarios where direct use of the loader is not feasible, applications can resort to explicit calls to 'LoadLibrary' and 'GetProcAddress'. This approach allows for targeted access to system functions.

3. Hunt-Pack Obfuscation Technique: The most notable technique discussed was the "Hunt-Pack" obfuscation method. It involves malicious code accessing specific memory locations in the Process Environment Block (PEB) and Thread Environment Block (TEB), which are located at a specific location within the FS register, to locate core DLL base addresses. This method often employs compact hashing of API string names using ROT13 or similar techniques for space-saving purposes. The malicious code then attempts to reverse the hashing algorithm to decode these strings back into recognizable forms.

The session emphasized the importance of understanding how obfuscation techniques work at a deeper level, such as recognizing patterns in memory access and hash manipulation, which are crucial for effective analysis and deobfuscation efforts.

Day 2, Part III – (Dropper.exe Analysis : The Hunt-Pack and its Resource Extraction)

The analysis of 'dropper.exe' continued with a focus on understanding where the output file buffer originated from and the subsequent steps taken by Windows APIs. The series of calls made included:

1. FindResource: This function attempts to locate the requested resource within the application's resources. It returns information about the resource, such as its name (e.g., "great_j"), size, and type if it was found.
2. LoadResource: If 'FindResource' is successful, this API call loads the identified resource into memory. This step confirms that the payload of interest is indeed present in application resources.
3. SizeofResource: Once loaded, 'SizeofResource' provides information about the size of the resource data. Knowing that the output file buffer was not directly accessible until after loading and sizing confirmed this assumption.
4. LockResource: Finally, 'LockResource' is crucial as it returns a pointer to the actual payload in memory. This indicates that the attacker had indeed placed their malicious code into application resources using these API calls.

The realization of the output file buffer being hidden within the application resources was a significant breakthrough and reinforced the concept of resource manipulation by attackers. It also highlighted the importance of understanding how Windows APIs interact with each other, particularly when dealing with dynamically loaded functions as documented in MSDN for better interpretation of arguments. This knowledge proved invaluable for correctly deducing the purpose of complex API calls within the context of application analysis.

Day 2, Part IV - (The '*this'* Pointer and Vtables in Object-Oriented Rev. Engg.)

The exploration of C++ object-oriented code delved into the significance of 'this' pointers, which serve as vital indicators within member functions. These pointers prominently appear before API calls and often reside within the ECX register. The 'this' pointer acts as a bridge between the function and the object it belongs to, allowing direct access to member variables through fixed offsets like 0x20 for age or 0x24 for a spouse pointer.

Another key aspect highlighted was the role of constructors in resource allocation. They leverage the powerful 'operator new' after pushing the size of the object onto the stack. This strategic use of memory management allowed for the manual reconstruction of complex structures, such as person_node (size 0x2C), which could be defined by correlating offsets to known attributes like name and age pointers, demonstrating a dynamic process in IDA for understanding these hierarchical relationships.

Day 2, Part V - [Advanced Object Analysis and Vtable Reconstruction]

This last session focused on mastering C++ object reversal techniques through a thorough understanding of Vtables and inheritance. Key insights emerged by tracing Vtable lookups, where the object's base address provides a pointer to the Vtable, which lists function pointers used for overridden methods.

One important lesson was in the identification of classes representing file and directory management that inherited from a common abstract base class. This inference could be made through observation of functions pointed to within the Vtables—for instance, 'CreateFileA' or 'CreateDirectory'. These examples significantly accelerated the reconstruction of the structure by using an IDC script to define multiple members based on observed offsets instead of tedious manual inputs. Furthermore, structural analysis was shown as a powerful tool for abstracted designs.