Analysis Report for: 1070AD568003F71387E34D0858975BDA.exe.c

Overall Functionality

This C code appears to be the decompiled output from a Go program, likely part of a larger malware binary. It heavily utilizes low-level functions and intrinsics, suggesting an attempt to obfuscate its true purpose. The code implements various functionalities related to memory manipulation, string comparison, CPU identification, and system calls, many of which are common in malware for tasks such as process injection, code execution, and anti-analysis techniques. The large number of functions with seemingly arbitrary names (`main_volumerememberincomeapril`, `main_smilesoup`, etc.) further points towards obfuscation. The presence of numerous functions related to Go's runtime indicates a Go binary was decompiled. The inclusion of Windows API functions suggests the malware targets Windows systems.

Function Summaries

Due to the obfuscation and the sheer volume of functions, providing detailed summaries for all 137 functions is impractical. However, I will analyze a representative subset to illustrate the code's nature:

* `internal_cpu_cpuid`: This function uses the `cpuid` instruction to gather information about the CPU. This is commonly used to detect CPU features and tailor malware behavior based on the hardware.

* `type__eq_*`: Several functions with this naming pattern perform equality checks on different Go data structures. These are likely crucial for runtime type checking within the Go program's structure.

* `runtime_memequal`, `memeqbody`: These functions compare memory regions, performing byte-by-byte comparisons, optimized using SIMD instructions. This is a fundamental operation often employed in malware for various malicious purposes, including comparing strings or data structures to determine next steps.

* `syscall_*`: Functions prefixed with `syscall` interact with the Windows operating system using system calls. These likely perform actions such as loading external libraries (`syscall_loadsystemlibrary`, `syscall_loadlibrary`), getting procedure addresses (`syscall_getprocaddress`), and making direct system calls (`syscall_Syscall`, etc.). This is vital for a malware's capability to interact with the underlying system.

* `main_*`: The functions prefixed with `main_` seem to be the actual core logic of the Go program. The numerous functions with obscure names are strongly indicative of obfuscation. Their complex interactions are difficult to ascertain without runtime context. The names themselves hint at potential functionality but remain misleading.

* `runtime_reflectcall`: This function indicates the potential use of reflection, a powerful technique in Go that could allow dynamic code execution. This is dangerous and highly suggestive of malicious intent.

* `runtime_gcWriteBarrier*`: These functions appear to manage Go's garbage collection write barriers. Their presence is typical in decompiled Go code, but they add to the complexity.

* `runtime_morestack`, `runtime_morestack_noctxt`: These functions handle stack growth within the Go runtime. Their use is expected, but their inclusion further suggests that this code is part of a larger Go program.

* `sync_atomic_*`: Atomic operations related to synchronization primitives are present, essential for thread safety within the Go program.

Control Flow

The control flow is generally complex, utilizing numerous loops and conditional statements:

* `cmpbody`: This function showcases the complex control flow and SIMD optimizations. It handles string comparisons differently depending on the length of the strings. The logic branches based on the length of data being compared, utilizing different comparison methods (byte-by-byte, 16-byte chunks with SIMD). It incorporates error handling for invalid comparison scenarios.

* `memeqbody`: This shows an adaptive method for memory comparison based on `byte_58DF83`. This is a common anti-analysis technique—the comparison implementation might vary based on some runtime condition or the presence of a debugger.

* `syscall_loadsystemlibrary`: Uses a combination of direct system calls and potentially Windows API functions through function pointers. This shows a common tactic used by malware for dynamically loading and executing malicious code.

* `runtime_reflectcall`: Contains a large switch statement to route the call depending on the number of arguments (size of the function call frame).

Data Structures

The code uses several key data structures, most of which are difficult to fully characterize without the original Go source code. However, some can be partially understood:

* `RTYPE`: This struct appears to describe Go's runtime type information.

* `__m128i`: This is a SIMD data type, indicating performance-oriented operations (likely memory manipulation or cryptographic operations).

* Custom structs with obfuscated names: Many structs are declared with names like `main_mountainpraise` and `main_comicintact`, demonstrating a deliberate effort to hide the purpose and structure of these data elements. These are very strongly indicative of obfuscation.

Malware Family Suggestion

Based on the observed functionality, this code is strongly suggestive of a sophisticated piece of malware, likely a downloader or dropper. The techniques used point towards a family that prioritizes obfuscation and evasive maneuvers to avoid detection. Pinpointing a specific malware family is not possible with only this code snippet; however, aspects like the use of system calls, function pointer usage, reflection, and extensive obfuscation techniques are typical of advanced persistent threats (APTs) or other highly targeted malware. The heavy use of Go, an increasingly popular language for malware authors, is another concerning aspect. Further analysis of the broader context (strings, network communication, etc.) within the larger malware sample would be needed for a conclusive identification.