Analysis Report for: 5E147A90B22DE4A1E740914D77E8930C.exe.c

Overall Functionality

This C code appears to be the output of a decompiler (Hex-Rays) acting on a Go program that likely performs some form of credential harvesting or unauthorized access. It heavily utilizes low-level system calls, string manipulation, and concurrency primitives, all pointing toward a potentially malicious purpose. The code interacts extensively with the Windows operating system (indicated by `windows.h` inclusion), suggesting it's designed specifically for Windows targets. The presence of numerous functions related to `syscall`, `os`, `os/signal`, `regexp`, and `time` further strengthens this hypothesis. The structure of many functions hints at operating system-level tasks, possibly including process manipulation, registry access, and network communication (although these specific functions are not explicitly named).

Function Summaries

The code contains a vast number of functions (over 200), many of which are likely related to Go's runtime environment. I will summarize only a selection of the most notable functions:

* `internal_cpu_cpuid`: Likely retrieves CPU information using the `cpuid` instruction.
* `type__eq_internal_cpu_option`, `type__eq______internal_cpu_option`: Equality checks for a custom data structure likely related to CPU options.
* `runtime_memequal`, `memeqbody`: Memory comparison functions, optimized for speed using SIMD instructions.
* `internal_bytealg_CountString`, `countbody`: String search functions, potentially optimized with SIMD.
* `syscall_Exit`: Terminates the process.
* `syscall_loadsystemlibrary`, `syscall_loadlibrary`, `syscall_getprocaddress`: Loads Windows DLLs and retrieves function addresses. Critical for loading additional malicious code.
* `runtime_cgocallbackg1_func3`: Go runtime function likely related to CGO (mixing Go and C code), potentially handling asynchronous operations.
* `runtime_getargp`: Retrieves a function argument. Suspiciously simple implementation.
* `runtime_printpointer`, `runtime_printuintptr`: Prints pointers and unsigned integers in hexadecimal format. Used for debugging or logging malicious actions.
* `sync_atomic_StorePointer`, `sync_atomic_CompareAndSwapPointer`: Atomic operations on pointers, essential for concurrent access to shared resources (likely used for locking/unlocking).
* `reflect_chanlen`, `internal_reflectlite_chanlen`, `reflect_maplen`, `internal_reflectlite_maplen`: Functions operating on channels and maps, suggesting a concurrent design for malicious activities.
* `main_main`: The main function of the program; orchestrates the main malicious actions.
* `main_runScan`: A key function responsible for a scan or attack likely involving credential verification or data exfiltration.
* `main_tryLogin`: Attempts to log in using harvested credentials.
* `main_loadPasswords`: Loads passwords from a specified file.

Control Flow

* `type__eq_internal_cpu_option`: This function performs a series of equality checks on various fields of a structure before using `runtime_memequal` to compare a memory region. This structure probably holds CPU feature information. This type of granular comparison is unusual, unless this information has significance in the malware's behavior.
* `main_runScan`: The control flow appears complex, likely due to concurrency and error handling. It loads passwords, then iterates (likely concurrently), attempting logins using `main_tryLogin`. It appears to involve logging successes to a file. It is very likely the malware's core operational logic.
* `main_tryLogin`: Uses a system call (`golang_org_x_sys_windows__ptr_LazyProc_Call`) which points towards potentially harmful actions. The return value indicates success or failure.
* `syscall_loadsystemlibrary`, `syscall_loadlibrary`, `syscall_getprocaddress`: These functions illustrate a common pattern in malware: loading external libraries to extend functionality, potentially through dynamic linking of further malicious code.

Data Structures

The code employs several data structures, many of which are obscured by the decompiler's representation. `RTYPE` structures seem to describe Go types, and several structures appear related to Go concurrency (e.g., channels, mutexes, wait groups). A `map` structure in the syscall functions stores Windows DLL functions, potentially for efficient retrieval. Note the large number of global variables including arrays which are very unusual, except for holding large amounts of data for either network or registry operations.

Malware Family Suggestion

Given the code's functionality (credential harvesting from a file, system calls, DLL loading, concurrent operations, and logging), the malware is likely a **credential stealer** or possibly a **remote access trojan (RAT)**. The complexity, and heavy use of concurrency and Windows-specific system calls suggest a sophisticated design probably aiming for persistence. The use of SIMD instructions in the string manipulation functions suggests an attempt to improve performance during scans. This program is not just a simple keylogger; it is more complex. Further analysis, especially of the un-decompiled code, is needed for a definitive determination.

Important Note: This analysis is based on decompiled code. The original Go source code would provide a much clearer understanding of the malware's capabilities and intentions. The analysis here highlights strong indicators of malicious intent, but a full assessment would require dynamic analysis in a sandboxed environment. Never run this code on a system unless it is in a controlled, safe environment.