

22:06 Mitigations are a reverser’s friend; or, Abusing XFG

by Aleksandar Nikolic

Control flow integrity protections, with its various implementations, have been the latest iteration of compiler mitigations for memory corruption exploits. They hope to make code reuse attacks more difficult or impossible. Implementation details vary, but all boil down to restricting possible valid targets of indirect calls. LLVM’s is called “Control Flow Integrity,” Grsecurity has “Reuse Attack Protector” and Microsoft’s is called “Control Flow Guard” (CFG).

The core idea behind Microsoft’s CFG is ensuring that function pointers can only point to valid function entry points before being used to perform a function call. The compiler inserts checks that, during runtime, inspect every indirect call instruction and terminate the process if the target isn’t a valid and known function start.

Putting aside the completeness or effectiveness of this mitigation, let’s ask whether we can glean some extra information about the code itself by the presence of these checks. As Deroko points out in Control Flow Guard Instrumentation,²¹ CFG mechanisms can serve as a way to hook all indirect calls in a binary without specifically looking for them in advance. They can also precisely identify function entry points, which is not always a trivial task.

ELIMINATES THE MIDDLEMAN!

AN INCREMENTAL RECORDER FOR ONLY \$3750.00

COMPLETE, READY TO OPERATE
Build it in (Rack Mount)
or carry it (Portable)

DIGI-DATA'S DIGITAL STEPPING RECORDER

STORE DATA AT ANY RATE — 0 TO 400 CHARACTERS PER SECOND

READ ON ANY IBM COMPATIBLE TRANSPORT

GENERATES FULLY COMPATIBLE IBM TAPE AUTOMATICALLY!

- Choice of 200 or optional 556 bits per inch
- Standard IBM type 10½" reels, ½" tape
- 7 track standard spacing
- ¾" inter record gap with longitudinal check character
- Lateral parity selectable—odd or even
- For additional information, phone or write

DDC DIGITAL STEPPING RECORDERS
DIGI-DATA CORPORATION DIGITAL DATA HANDLING EQUIPMENT

DIGI-DATA CORPORATION
Main Office: 4315 Baltimore Ave. • Bladensburg, Md. 20710 • (301) 277-9378
Western Regional Office: 4341 W. Commonwealth Ave. • Fullerton, Cal. 92633 • (213) 941-3182

With the release of Windows 11, Microsoft is introducing another iteration of control flow integrity mitigation called “eXtended Flow Guard” or XFG. In short, it further restricts targets of indirect calls to not only valid function entry points, but to a subset of functions that have a particular signature consisting of return value type, number and types of parameters and other function properties.

Surely, this added metadata can somehow aid us in our reverse engineering process. To see how, we’ll need to understand the implementation details.

What is XFG and how it works

Extended Flow Guard is introduced as a compiler extension that can be enabled via `/guard:xfg` switch that is available in MS’s C and C++ compilers since at least the 19.27.29112 version of Visual Studio 2019. It hasn’t seen full support or much public use until release of Windows 11. Consider an example:

```
int test(){
    return 0;
}

int (*cfgTest[1])() = {test};

int main(){
    cfgTest[0]();
}
```

This code has a simple function pointer array `cfgTest` and makes a call to `test` using that function pointer. If compiled with `cl /Zi /guard:xfg simple.c` its assembly looks a little odd.

```
1 sub     rsp, 38h
2 mov     eax, 8
3 imul   rax, 0
4 lea    rcx, cfgTest
5 mov    rax, [rcx+rax]
6 mov    [rsp+38h+var_18], rax
7 mov    r10, 0D30527475E523070h
8 mov    rax, [rsp+38h+var_18]
9 call   cs:__guard_xfg_dispatch_icall_fptr
10 xor    eax, eax
11 add    rsp, 38h
12 retn
```

²¹[unzip pocorgtfo22.pdf cfghook.zip](#)

This is some peculiar code. There is no indirect call to function `test`, rather there's a call to `__guard_xfg_dispatch_icall_fptr` with certain arguments. The function pointer is actually saved in `rax` and an odd-looking constant is moved into `r10` before `__guard_xfg_dispatch_icall_fptr` is called. This odd-looking constant is what we will call an XFG hash. Interestingly, if we take a look at `test` function's prologue on page 25, we'll see (almost) the same data.

Long story short, before invoking the target function, `__guard_xfg_dispatch_icall_fptr` will check that the hash in `r10` matches the hash located right before the function. If they don't match, process is terminated.²²

This ensures that only legal target functions can be executed at this particular indirect function call. The next obvious question is: how is this function hash derived? That brings us to the core idea behind XFG.

If we think about it, no matter how an indirect call instruction happens to be generated by the compiler, several things are true for all the possible, valid, target functions in a valid program. All possible target functions must have the same number of arguments, the same calling convention, same argument types, same return value type and so on. Even if the compiler doesn't know of all possible target functions in advance, it must know all of these facts about those targets. It can, then, generate a unique representation of those facts when it encounters an indirect function call. On the other hand, for every function that could be a possible target for indirect call, the same unique representation can be calculated and those two can be compared during runtime.

This unique representation of function prototype information is what constitutes an XFG hash.

How is an XFG hash generated?

Francisco Falcon over at Quarkslab has already done the hard work of reverse engineering most of XFG internals. Their extended writeup provides a number of examples.²³ XFG hash generation happens in the `cl.exe` compiler's frontend `cl.dll` and revolves around gathering function prototype information and using the SHA256 hashing algorithm on it while following certain rules. A list of function prop-

erties that figure into the XFG hash is (as far as C code is concerned at least) as follows:

- number of arguments
- the types of individual arguments
- type of return value
- whether the function is variadic or not
- the calling convention

When preparing to calculate the hash, each of these is represented in a specific way. Some are simple constants, while others have more structure and are often recursively defined. For example, the number of arguments is just represented as a 32-bit integer, the calling convention appears to be a 16-bit constant, and variadic is one byte boolean. Return value and argument types, on the other hand, are more complicated.

Those consist of values specifying type qualifiers (`const`, `volatile`), type groups (primitives, pointers, structs/unions/enums), and values according to the type group. Calculating values for primitive types are the simplest and are just a table lookup:

"void"	:0xe,
"char"	:0x1,
"signed char"	:0x1,
"unsigned char"	:0x1,
"__int8"	:0x1,
"char8_t"	:0x1,
"__int16"	:0x6,
"short int"	:0x6,
"unsigned short int"	:0x86,
"float"	:0x11,
"int"	:0x7,
"__int32"	:0x7,
"unsigned int"	:0x87,
"long int"	:0x10,
"unsigned long int"	:0x8a,
"double"	:0x12,
"__int64"	:0x8,
"long double"	:0x12,
"long long int"	:0x8,
"unsigned long long int"	:0x88,
"unsigned long long"	:0x88,

Notice that there are several distinct primitive types that have the same value. Structs, unions, and enums are treated the same, and their actual (verbatim text) names are included as part of a hash calculation.

²²A great in-depth description from Connor McGar is available as *Exploit Development: Between a Rock and a (Xtended Flow) Guard Place: Examining XFG*.

²³See *How the MSVC Compiler Generates XFG Function Prototype Hashes* by Francisco Falcon.

```

.text:0000000140001008          dq  0D30527475E523071h
.text:0000000140001010
.text:0000000140001010 ; ===== S U B R O U T I N E =====
.text:0000000140001010
.text:0000000140001010
.text:0000000140001010 ; int test(...)
.text:0000000140001010 test      proc near          ; DATA XREF: .rdata:__guard_fids_table
.text:0000000140001010                                ; .data:cfgTest
.text:0000000140001010                                xor     eax, eax
.text:0000000140001012                                retn
.text:0000000140001012 test      endp

```

Figure 4: `test` Function’s Prologue

Pointers of any kind are the most complicated, as their value is the hash of the type they point to, requiring recursive evaluation.

This can look a bit confusing and — although it’s covered in great detail in the referenced Quarkslab article — we’ll illustrate the process with the simplest example. We’ll add a void pointer as an argument to `test` from before:

```
int test(void *arg);
```

First, there’s only a single argument to this function, so we will append “\x01\x00\x00\x00” to our data to be hashed (`data0`). Second, we need to consider function arguments, calculate their hashes, and append them to data to be hashed. There is only one argument and it’s a pointer without qualifiers. Starting a new hash (`data1`), we append “\x00” for qualifiers, “\x03” for type group but then we need to consider the type of pointer and calculate that hash separately. Starting yet another hash calculation (`data2`), we append “\x00” for qualifiers, “\x01” for type group and finally “\x0e” for primitive type. Calculate the SHA256 of `data2` and append its first 8 bytes to `data1` that completes necessary data for calculating first argument hash. Hash `data1` and append the first 8 bytes to `data0`. That concludes the argument part of the hash. Next is whether the function is variadic, so we append “\x00” and what the calling convention is, which defaults to just “\x01”. The last segment is the return value type which is an integer primitive, so it’s simply “\x00” for qualifiers, “\x01” for type group and finally 0x7 for a primitive type. The hash of that is appended to `data0`.

Putting that together gives us the following, with all SHA256 results truncated to the first eight bytes.

```

sha256("\x01\x00\x00\x00"
+sha256("\x00\x03"+sha256("\x00\x01\x0e"))
+"\x00"+" \x01"+sha256("\x00\x01\x07"))

```

After some final transformations, the result of the operation is the “719a5e103606e1b2” value that appears before the `test` function in the binary.

An implementation of this algorithm, in Python, that parses a given C function prototype and generates its corresponding hash can be found as an attachment.²⁴

INFORMATION SYSTEMS SCIENTISTS AND ENGINEERS

Bendix Research Laboratories has excellent career opportunities for B.S., M.S., and Ph.D. graduates with 2 to 15 years experience in one or more of the following key areas:

- **ARTIFICIAL INTELLIGENCE**, pattern recognition, trainable systems, adaptive logic.
- **COHERENT OPTICAL PROCESSING**, spatial filtering, optical correlation, electro-optical systems and components.
- **DIGITAL COMPUTER DESIGN**, systems analysis, logic and circuit design, real-time computer control.
- **ANALOG COMPUTERS AND SYSTEMS**, information theory, control theory, circuit and servo analysis, correlation techniques.
- **REAL-TIME COMPUTER PROGRAMMING**, mathematical analysis, scientific computing.

Assignments involve research and development in automatic extraction of information from photographic records, image processing and analysis, adaptive and trainable control systems, digital and hybrid computing techniques, and advanced real-time computer control applications.

Interested individuals are invited to call collect or to send a resume to our Personnel Director.

Research Laboratories Division
Southfield, Michigan • (313) 353-3500

An equal opportunity employer



²⁴unzip -p pocorgtfo22.pdf xfg-scripts-args.tgz | tar -xzvf- gen_hash_from_ast.py

Using XFG to resolve indirect jumps

Now that we know how XFG works, we can consider how it can be of use as a reverse engineering aid.

The first, and most obvious idea is that it can reduce the uncertainty of analyzing indirect calls. Since all indirect calls in an XFG-protected binary will inevitably be dispatched through `__guard_xfg_dispatch_icall_fptr` that must match callsite's hash and target function's hash, it should be possible to enumerate all possible targets completely statically (assuming all possible linked code is known/available for analysis).

Let's illustrate this with an example. Throughout the rest of the article, we'll use `ntdll.dll` binary from Windows 11 for illustrations and testing. If we go to function `LdrQueryProcessModuleInformationEx` and take a look at the following piece of assembly:

```
18000174e 488d04bf
  lea rax, [rdi+rdi*4]
180001752 49ba7048da56963e...
  mov r10, 0x85f13e9656da4870
18000175c 498b44c118
  mov rax, qword [r9+rax*8+0x18]
180001761 ff15a9181900
  call qword
    [rel __guard_xfg_dispatch_icall_fptr]
    {j_sub_1800aa130}
180001767 4c8d0df2b71200
  lea r9, [rel data_18012cf60]
```

While we don't know without debugging what possible target this XFG dispatch call might have, we can see that its hash must be `0x85f13e9656da4871` (the 1 is added at the end of the supplied hash by dispatcher). If we search the binary for functions that have this XFG hash, we'll find many results: `LdrQueryModuleInfoLocalLoaderUnlock`, `LdrShutdownThread`, `LdrShutdownProcess`, `RtlDetectHeapLeaks`, `TpTrimPools`, `RtlCleanUpTEBLangLists`, `RtlFreeThreadActivationContextStack`, `LdrProcessInitializationComplete`, `RtlFlushHeaps`, `RtlReleasePebLock`, `RtlAcquirePebLock`, `LdrFastFailInLoaderCallout`, ...

Obviously, from the function names, not all of these make sense as possible targets for this indirect call because of their differing semantics, but there's a good chance that all with `Ldr` prefix are actual possible targets.

Why are there so many hash hits that are unlikely to be real targets? It's probable that the tar-

get function prototype in this case is very simple, and matches many other functions. In fact, hash `0x85f13e9656da4871` represents the simplest possible case of `'void fname()'`. As another example, the `CppCallbackEpilog` function has the following indirect call:

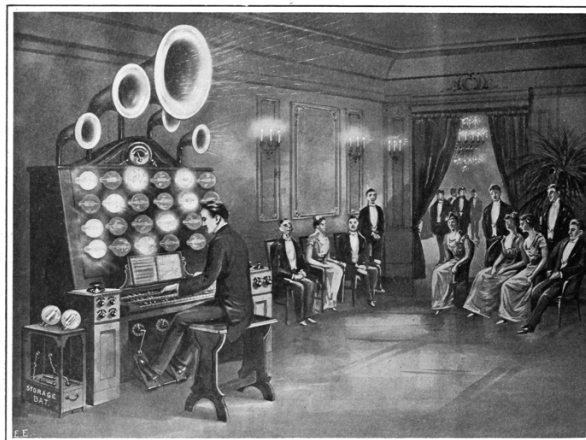
```
18001766e 488b8eb8000000
  mov rcx, qword [rsi+0xb8]
180017675 4c89aeb8000000
  mov qword [rsi+0xb8], r13 {0x0}
18001767c 488b4108
  mov rax, qword [rcx+0x8]
180017680 49ba70125178f527...
  mov r10, 0xa6d127f578511270
18001768a 488b4008
  mov rax, qword [rax+0x8]
18001768e ff157cb91700
  call qword
    [rel __guard_xfg_dispatch_icall_fptr]
    {j_sub_1800aa130}
```

Looking up the target hash, `0xa6d127f578511271`, in the binary yields: `TppSimpleFree`, `TppWorkpFree`, `TppAlpcpCallbackEpilog`, `TppJobpCallbackEpilog`, `TppFreeWait`, `TppTimerpFree`, `TppIopFree`, `TppAlpcpFree`, `TppJobpFree`, `TppWorkCancelPendingCallbacks`, `TppIopCancelPendingCallbacks`.

All of these look like possible real targets given their context.

So while not completely precise, this simple static analysis that relies on XFG hashes definitely sheds some light on indirect calls that might otherwise remain completely unresolved.

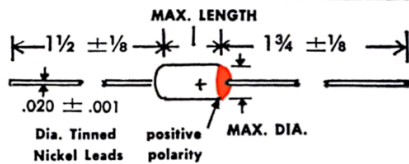
Attached is a Binary Ninja plugin that annotates indirect calls with information gained by XFG analysis.²⁵



²⁵`unzip -p pocorgtfo22.pdf xfg-scripts-args.tgz | tar -xzvf- xfg_analyzer.py`

ECONOTAN

SOLID TANTALUM CAPACITORS



METAL CASE			POLYESTER SLEEVE INSULATION		
SERIES	MAX. DIA.	MAX. LENGTH	SERIES	MAX. DIA.	MAX. LENGTH
CT	.090	.240	CT	.095	.260
CM	.128	.300	CM	.133	.320
CL	.175	.325	CL	.180	.325

SMALL SIZE
WITH MAXIMUM CAPACITANCE

LONG SHELF LIFE
ALL DRY CONSTRUCTION

LOW COST
WELL BELOW 20¢ EACH
IN PRODUCTION QUANTITIES

AVAILABLE RATINGS
STANDARD CAPACITANCE TOLERANCE ±20%

PART NUMBER	CAP MFD	VV DC	MAX DF	MAX IL
CT683	.068	25	.08	1.0
CT104	.10	25	.08	1.0
CT154	.15	25	.08	1.0
CT224	.22	25	.08	1.0
CT334	.33	25	.08	1.0
CT474	.47	25	.08	1.0
CT684	.68	25	.08	1.0
CT105	1.0	25	.08	1.0
CT155	1.5	20	.08	1.0
CT225	2.2	15	.08	1.0
CT335	3.3	10	.08	1.0
CT475	4.7	6	.10	1.0
CT685	6.8	4	.12	1.0
CT106	10.0	2	.12	1.0

PART NUMBER	CAP MFD	VV DC	MAX DF	MAX IL
CM155	1.5	25	.08	1.0
CM225	2.2	25	.08	1.0
CM335	3.3	25	.08	1.0
CM475	4.7	20	.08	1.0
CM685	6.8	15	.08	1.0
CM106	10.0	10	.08	1.0
CM156	15.0	6	.10	1.0
CM226	22.0	4	.12	1.0
CM336	33.0	2	.12	1.0

PART NUMBER	CAP MFD	VV DC	MAX DF	MAX IL
CL475	4.7	25	.08	2.0
CL685	6.8	25	.08	2.0
CL106	10.0	20	.08	2.0
CL156	15.0	15	.08	2.0
CL226	22.0	10	.08	2.0
CL336	33.0	6	.10	2.0
CL476	47.0	4	.12	2.0
CL686	68.0	2	.12	2.0

Visit us at
BOOTH 2A-48
IEEE Show

TECHNICAL BULLETIN AVAILABLE ON REQUEST

COMPONENTS, INC.

SMITH STREET, BIDDEFORD, MAINE
PHONE A.C. — 207-284-5956
PLANTS AT BIDDEFORD AND KENNEBUNK

Brute forcing XFG hashes for function prototype recovery

Another, more involved, idea stems from the fact that XFG hashes aren't random and actually encode function prototypes. Surely, there would be a way to recover at least some of that information and make use of it.

While it is not possible to reverse the hash back to function prototype directly, it is perfectly feasible to precompute a lookup table for all possible function prototypes (up to certain number of arguments). If we ignore structs, unions and enums for a second, there are only a fairly small number of primitive types. In fact, if we remove the duplicates, there's a total of only 12 primitive types (with distinct values as far as XFG generation is concerned). Adding in type qualifiers (const, volatile) and pointers, a bit of simple combinatorics tells us that total number of all possible function prototypes is roughly $(12 * 3)^{num_args} + 1$.

This gets big very fast as we increase the number of arguments, but the whole list is precomputed in minutes for functions up to three arguments.

```

import sys
2 import itertools
from jinja2 import Template
4 types = ["void", "char", "short int",
           "unsigned short int", "float",
6          "int", "unsigned int", "long int",
           "unsigned long int", "double",
8          "long long int", "unsigned long long"]
# add all types as pointers
10 types += [x + " *" for x in types]
# and as consts
12 types += ["const " + x for x in types]
# and as volatiles
14 types += ["volatile " + x for x in types]

16 j2_template = Template("""
    {{ret_type}} fname( {%- for param_type in
        param_types -%} {{param_type}} arg{{loop
        .index}}{ " , " if not loop.last }} {%-
        endfor -%});
18 """)

20 max_func_params = 3
f = open(sys.argv[1], "w")
22 i = 0
for ret_type in types:
24     for pn in range(4, max_func_params+1):
         for c in itertools.product(types,
26                                     repeat=pn):
             f.write(j2_template.render({"ret_type":
28             : ret_type, "param_types": c}))
         i+=1
f.close()

```

This code uses a jinja2 template to generate an exhaustive list of all possible function prototypes starting with given primitive types. These generated prototypes can then be fed into the hash generation algorithm to compile a lookup table.

So, does this work? We'll test this on `ntdll.dll` again. This particular version of the DLL has a total of 1564 functions that have an XFG hash associated with them. Out of those 1564, there are a total of 995 unique XFG hashes. After lookups, this simple matching has identified function prototypes for 131 unique hashes, corresponding to a total of 294 functions!

By simply precomputing all possible function prototypes up to three parameters (using nothing target specific, only primitive types) we were able to recover precise function prototypes for about 13% of unique hashes in `ntdll.dll`. Figure 5 has some examples.

The proof-of-concept works, but there are a couple of reasons why we didn't get a higher hit rate. First and most obvious is that many functions simply have more than three arguments, but even bigger factor is the fact that `ntdll.dll` code heavily relies on use of structures, enums, and structure pointers. Since hashes for struct, union, and enum types include their names directly, straight up brute forcing isn't practical, but seeding certain (domain specific) names would greatly increase the hit rate. XFG hash calculation implementation supports structs in function prototypes, and since structs, enums, and unions are treated the same, all we need to do to add struct names is to expand the list of primitive types. Adding `struct in_addr` to list of primitive types leads to following result:

```
7139d252a1b76de8 char *func(
    const struct in_addr *arg1, char *s)
```

This calculated hash matches the XFG hash for `RtlIpv4AddressToStringA`. By adding target specific, commonly used, structs to prototype generation we can greatly increase the number of found hashes at the expense of a larger lookup table. Since structures and other type information are sometimes publicly available even if function prototypes are not, this allows for very precise function prototype recovery.

How do we know that these results are actually correct? Let's take another look at an example where we do know the function prototype. Function

'`RtlSetUserValueHeap`' has four arguments. Binary Ninja guesses its prototype to be:

```
void* const* RtlSetUserValueHeap(
    int64_t arg1, int32_t arg2,
    int64_t arg3, int64_t arg4);
```

Similarly, IDA guesses:

```
char __fastcall RtlSetUserValueHeap(
    __int64 a1, unsigned int a2,
    __int64 a3, __int64 a4)
```

This function's XFG hash is `0xc76c3600585a-f171` and a lookup reveals the following function prototype:

```
char RtlSetUserValueHeap(
    void *arg1, unsigned long int arg2,
    void *arg3, void *arg4);
```

Notice how both Binary Ninja and IDA cannot know that some of the arguments are pointers. This simple fact adds a lot of information that greatly aids further function analysis and decompilation. And what about correctness? While source for '`RtlSetUserValueHeap`' isn't available, it is reimplemented in ReactOS where its function prototype is:

```
BOOLEAN
NTAPI
RtlSetUserValueHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _In_ PVOID BaseAddress,
    _In_ PVOID UserValue
);
```

While the prototype gathered from XFG analysis lacks some extra annotations, the types themselves match precisely!

In Conclusion

Even though mitigations like XFG pose a real challenge when it comes to exploitation, it sometimes pays off to take a step back and consider the possible side effects that can be handy in other ways. We've shown that a very simple lookup table can recover a treasure trove of information that can be helpful when reverse engineering an XFG-protected binary. As XFG adoption spreads to code other than Microsoft's, this can definitely lead to some interesting discoveries.

```

char RtlGetSecurityDescriptorRMControl(void *arg1, char *arg2);
unsigned long int RtlNumberOfSetBitsUlongPtr(unsigned long long int arg1);
char RtlEqualWnfChangeStamps(unsigned long int arg1, unsigned long int arg2);
unsigned long int RtlSetProxiedProcessId(unsigned long int arg1);
void RtlWnfDllUnloadCallback(void *arg1);
void *memchr(const void *arg1, int arg2, unsigned long long arg3);
char *strchr(const char *arg1, int arg2);
unsigned long long strcspn(const char *arg1, const char *arg2);
unsigned long long strlen(const char *arg1, unsigned long long arg2);
char *strpbrk(const char *arg1, const char *arg2);
char *strrchr(const char *arg1, int arg2);
unsigned long long strspn(const char *arg1, const char *arg2);
char *strstr(const char *arg1, const char *arg2);
int tolower(int arg1);
int WinSqmCommonDatapointSetDWORD64(
    unsigned long int arg1, unsigned long long arg2, unsigned long int arg3);
int WinSqmCommonDatapointSetString(
    unsigned long int arg1, const unsigned short int *arg2, unsigned long int arg3);
int WinSqmGetInstrumentationProperty(
    const unsigned short int *arg1, const unsigned short int *arg2,
    unsigned short int *arg3, unsigned long int *arg4);
int WinSqmIsOptedInEx(unsigned long int arg1);
void AlpcGetCompletionListLastMessageInformation(
    void *arg1, unsigned long int *arg2, unsigned long int *arg3);
unsigned long int DbgPrompt(const char *arg1, char *arg2, unsigned long int arg3);
char RtlQueryProcessPlaceholderCompatibilityMode();
char RtlSetProcessPlaceholderCompatibilityMode(char arg1);
char RtlIsNonEmptyDirectoryReparsePointAllowed(unsigned long int arg1);
char RtlIsZeroMemory(void *arg1, unsigned long long arg2);
unsigned short int RtlLogStackBackTrace();
void *RtlLogStackTrace(unsigned long int arg1);
void RtlReleaseStackTrace(void *arg1);

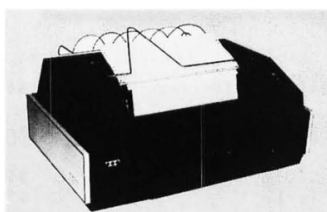
```

Figure 5: Example Prototypes from ntdll.dll

TELETYPES®


IMMEDIATE DELIVERY

MODEL 40 300 LPM PRINTERS



- Mechanism or complete assembly
- 80-column friction feed
- 80-column tractor feed
- 132-column tractor feed

MODEL 43 TERMINALS



- 4310 RO (Receive Only)
- 4320 KSR (Keyboard Send-Receive)
- 4340 BSR (Buffered Send-Receive)

INTERFACES

- EIA-RS232
- Simplified EIA-like interface
- Standard serial interface
- Parallel device interface

INTERFACES

- TTL Serial
- EIA RS232 or DC20 to 60ma
- 103-type built-in modem

FEDERAL Communications Corporation

11126 Shady Trail, Dallas, Texas 75229, (214) 620-0644,
TELEX 732211 TWX 910-860-5529