

Forum:[Hack Forums](#)

Thread: [C++ Payload Blaster](#)

nucleo_type09 - April 12, 2024, 4:05 pm

c++ Payload Blaster, take a raw Buffer, output a JSON like block...

you can find the code here: https://github.com/nucleoType98/Payload_Blaster.git

----- Some Demo Here -----

```
Code #include "../include/blaster.cpp"
```

```
using namespace Payload; // Namespace Payload, or use directly Payload::Blaster
```

```
int main() {
```

```
    // declare a key for the block, this will be the key that represents a block
    Payload::tDataBlockKey key = 42323;
```

```
    /**
     * declare a value which in real world will be the actual raw buffer data as character array(string)
    format.
    **/
    Payload::tDataBlockValue value = "{key_1: \"some value\\", key_2: \"some value 2\\", key_3: \"some
    value 3\\\"}";
```

```
    // creating unit, intersect key with values
    tKeyValueUnit u{key, value};
```

```
    /**
     * create a Payload instance with an initial payload block, the initial block is optional, but if you do
    not set it here,
     * you must execute payload_allocate(...) to allocate one before last operation (Blast())
    **/
    Payload::Blaster B1(u);
```

```
    /**
     * Required Step: register operation scheme.
     * Note that im setting one of the keys(key_1) to a wrong value(XXXkey_1) on purpose, to show
    how the operation fails if some keys go missing.
    **/
    B1.register_scheme("logs", {"XXXkey_1", "key_2", "key_3"});           // <-- Intersects with "u"
```

```
    /**
     * default Blast mode is set to "STRICT", meaning if even only one key is missing from the target,
    the operation brutally fails.
     * If you want to allow partial results, then set the argument value to SCAN_MODE::LOOSE, which
    allows for loosed values, but returns misplaced record position.
    **/
    B1.Blast();
```

```

/**
 * Collect Final block, collector will return a structure containing;
 * - Data -> an unordered_map type containing field/value pairs of final result
 * - Size -> the number of entries returned (key-value) pair block size
 * - State -> the state of the operation, true if success, false if failure
 */
Payload::CollectType collector = B1.collect();

std::cout << "----- SCHEME LOGS -----" << std::endl;

/**
 * Display the result.
 * NOTE: this operation performs a direct failure, as we provided a wrong key within the
register_scheme() call
 * and did NOT set Blast() call SCAN_MODE argument to SCAN_MODE::LOOSE in order to allow
loosed result, if want to
 * return partial result even if key "x" is missing, use Blast(SCAN_MODE::LOOSE).
 * The Operation Will Choke!
 */
if (collector.State)
    for (auto &frame : collector.Data)
        std::cout << frame.first << " = " << frame.second << std::endl;

/**
 * Reallocate "logs" scheme, this time using the right field ids.
 * NOTE: since "logs" is already allocated, the operation requires flag(SCHEME_ALLOC_MODE)
set to OVERWRITE_IF_FOUND to overwrite the
 * current "logs" scheme, otherwise will not have any effect.
 */
B1.register_scheme("logs", {"key_1", "key_2", "key_3"},
SCHEME_ALLOC_MODE::OVERWRITE_IF_FOUND);

B1.Blast();

collector = B1.collect();

/**
 * Display the result.
 * This Operation will return Success as i provided the right key/value pair.
 */
if (collector.State)
    for (auto &frame : collector.Data)
        std::cout << frame.first << " = " << frame.second << std::endl;

// lock the data
B1.set_lock();

// remove the lock, extract data
B1.drop_lock();

```

```
    return 0;
}
```

Without all those crappy comments...

```
Code #include "../include/blaster.cpp"
```

```
using namespace Payload; // Namespace Payload, or use directly Payload::Blaster
```

```
int main() {
```

```
    Payload::tDataBlockKey key = 42323;
    Payload::tDataBlockValue value = "{key_1: \"some value\", key_2: \"some value 2\", key_3: \"some
value 3\"}";
    tKeyValueUnit u{key, value};
    Payload::Blaster B1;
    B1.register_scheme("logs", {"XXXkey_1", "key_2", "key_3"});
    B1.payload_allocate(u);
    B1.Blast();
    Payload::CollectType collector = B1.collect();
    B1.set_lock();
    B1.drop_lock();
    return 0;
}
```

Hackers - April 14, 2024, 10:20 am

I am not 100% sure if I understand what this is doing.

What ultimately is used for the json keys? Is it defining the functions, vars, types, etc? And can it then convert back from JSON too?

Looks interesting

nucleo_type09 - April 14, 2024, 11:10 am

(Apr 14, 2024 10:20 AM)nodes Wrote: [\[CITE\]](#) I am not 100% sure if I understand what this is doing.

What ultimately is used for the json keys? Is it defining the functions, vars, types, etc? And can it then convert back from JSON too?

Looks interesting

well yeah, i designed it originally for my other codebase which is a unix IRC, this was not a class by itself but raw integration within the IRC program, but as it was pretty big, i decided to convert it into a class so i can post it on github and maybe on other projects i could just clone it and use it.
so that's how it works...

first you create an instance of Blaster(BTW i called so because essentially it blasts the raw buffer into segments)

```
Code Payload::Blaster Blaster1;
```

then you define a scheme you expect to receive, for example if you expecting the buffer to contain keys[key1, key2]

```
Code Blaster1.register_scheme("scheme_name", {"field1", "field2", ...});
```

when you define a new scheme that scheme will automatically set itself to the current scheme, but if you define multiple schemes in a shot, you can then decide to switch to anyone like this:

```
Code Blaster1.use_scheme("scheme_name");
```

after that, you can allocate any payload you want by providing a key as reference to that specific payload block so you can reference it later for locking/unlocking/erasing etc.. operations..

```
Code Payload::tDataBlockKey key = 42323;
Payload::tDataBlockValue buffer = "{key: \"some value\", key2: \"some value 2\"}";
tKeyValueUnit payload_unit{key, value};
Blaster1.payload_allocate(payload_unit);
then you can blast it using...
```

```
Code Blaster1.Blast(); // this will explode the payload into segments you defined as scheme fields
```

finally, you can collect the parsed buffer just like this..

```
Code auto collect = Blaster1.collect(); // here you got an unordered_map with key/value pairs from the buffer
```

you can have a look at the git repo if you want, i provided the full documentation about the code at https://github.com/nucleoType98/Payload_Blaster.git

What do you think? it is a kind of JSON parser but with more features built into it, specifically designed to work with networking, for new payloads that you might need to parse or analyse, if also interests you, i compiled it using g++ compiler with additional -fsanitize flags and valgrind to check mem-leaks...

??

Hackers - April 15, 2024, 7:12 am

(Apr 14, 2024 11:10 AM)nucleo_type09 Wrote: [\[CITE\]](#) well yeah, i designed it originally for my other codebase which is a unix IRC, this was not a class by itself but raw integration within the IRC program, but as it was pretty big, i decided to convert it into a class so i can post it on github and maybe on other projects i could just clone it and use it.
so that's how it works...

first you create an instance of Blaster(BTW i called so because essentially it blasts the raw buffer into segments)

```
Code Payload::Blaster Blaster1;
then you define a scheme you expect to receive, for example if you expecting the buffer to contain keys[key1, key2]
```

```
Code Blaster1.register_scheme("scheme_name", {"field1", "field2", ...});
when you define a new scheme that scheme will automatically set itself to the current scheme, but if you define multiple schemes in a shot, you can then decide to switch to anyone like this:
```

```
Code Blaster1.use_scheme("scheme_name");
```

after that, you can allocate any payload you want by providing a key as reference to that specific payload block so you can reference it later for locking/unlocking/erasing etc.. operations..

```
Code Payload::tDataBlockKey key = 42323;
```

```
Payload::tDataBlockValue buffer = "{key: \"some value\", key2: \"some value 2\"}";  
tKeyValueUnit payload_unit{key, value};  
Blaster1.payload_allocate(payload_unit);  
then you can blast it using...
```

Code `Blaster1.Blast();` // this will explode the payload into segments you defined as scheme fields

finally, you can collect the parsed buffer just like this..

Code `auto collect = Blaster1.collect();` // here you got an `unordered_map` with key/value pairs from the buffer

you can have a look at the git repo if you want, i provided the full documentation about the code at https://github.com/nucleoType98/Payload_Blaster.git

What do you think? it is a kind of JSON parser but with more features built into it, specifically designed to work with networking, for new payloads that you might need to parse or analyse, if also interests you, i compiled it using g++ compiler with additional `-fsanitizer` flags and `valgrind` to check mem-leaks...

??

Ah, thank you for providing this much detail!!

I thought for some reason you were converting binary data/programs into JSON, like deconstructing a PE and then using some sort of thing to convert the defined funcs, variables, etc into JSON that can then be reconstructed later from JSON for all headers, sections, etc. I think I got thrown off by 'payload' blaster and the buffer

This makes sense though, I still need to look at your code more and try it out, but I think I was confused in my last reply

nucleo_type09 - April 15, 2024, 8:26 am

(Apr 15, 2024 07:12 AM)nodes Wrote: [CITE](#) Ah, thank you for providing this much detail!!

I thought for some reason you were converting binary data/programs into JSON, like deconstructing a PE and then using some sort of thing to convert the defined funcs, variables, etc into JSON that can then be reconstructed later from JSON for all headers, sections, etc. I think I got thrown off by 'payload' blaster and the buffer

This makes sense though, I still need to look at your code more and try it out, but I think I was confused in my last reply

haha, is it that confusing? should i rename it `jsonBlaster` instead? the fact is i called it `payloadBlaster` bcz the initial integration was within another unix socket program that received payloads with this class, that's why i called it `payloadBlaster` haha